

This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images,
please do not report the images to the
Image Problem Mailbox.**

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁵:

G06F 15/32

A2

(11) International Publication Number:

WO 94/23385

(43) International Publication Date:

13 October 1994 (13.10.94)

(21) International Application Number: PCT/GB94/00677

(22) International Filing Date: 30 March 1994 (30.03.94)

(30) Priority Data:

040,301

30 March 1993 (30.03.93)

US

100,747

30 July 1993 (30.07.93)

US

(71)(72) Applicants and Inventors: LEWIS, Adrian, Stafford [GB/ES]; Federico Garcia Lorca 17-5-B, E-07014 Palma (ES). KNOWLES, Gregory, Percy [AU/ES]; Calle Menorca 18-2-B, E-07011 Palma (ES).

(74) Agent: JONES, Ian; W.P. Thompson & Co., Celcon House, 289-293 High Holborn, London WC1V 7HU (GB).

(81) Designated States: AT, AU, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, ES, FI, GB, HU, JP, KP, KR, KZ, LK, LU, LV, MG, MN, MW, NL, NO, NZ, PL, PT, RO, RU, SD, SE, SI, SK, TT, UA, UZ, VN, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

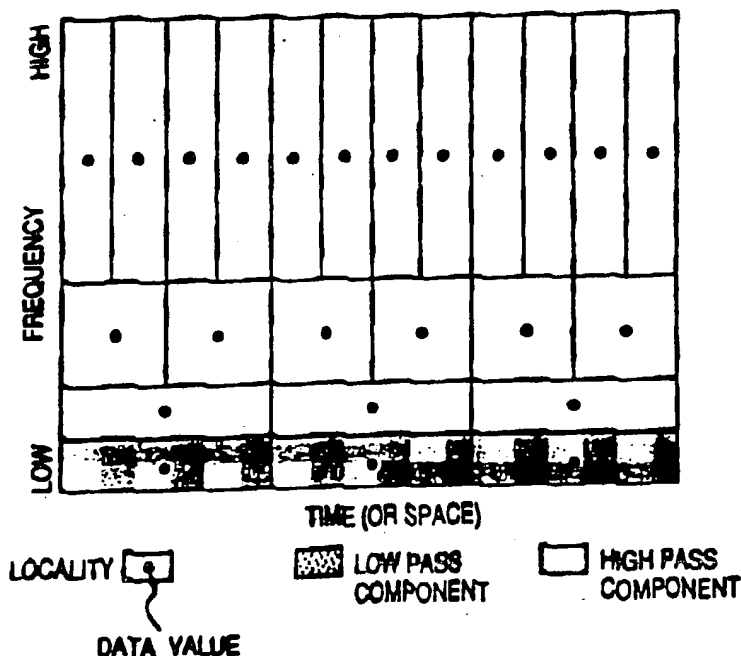
Published

Without international search report and to be republished upon receipt of that report.

(54) Title: DATA COMPRESSION AND DECOMPRESSION

(57) Abstract

A compression and decompression method uses a wavelet decomposition, frequency based tree encoding, tree based motion encoding, frequency weighted quantization, Huffman encoding, and/or tree based activity estimation for bit rate control. Forward and inverse quasi-perfect reconstruction transforms are used to generate the wavelet decomposition and to reconstruct data values close to the original data values. The forward and inverse quasi-perfect reconstruction transforms utilize special filters at the boundaries of the data being transformed and/or inverse transformed. Structures and methods are disclosed for traversing wavelet decompositions. Methods are disclosed for increasing software execution speed in the decompression of video. Fixed or variable length tokens are included in a compressed data stream to indicate changes in encoding methods used to generate the compressed data stream.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	GB	United Kingdom	MR	Mauritania
AU	Australia	GE	Georgia	MW	Malawi
BB	Barbados	GN	Guinea	NE	Niger
BE	Belgium	GR	Greece	NL	Netherlands
BF	Burkina Faso	HU	Hungary	NO	Norway
BG	Bulgaria	IE	Ireland	NZ	New Zealand
BJ	Benin	IT	Italy	PL	Poland
BR	Brazil	JP	Japan	PT	Portugal
BY	Belarus	KE	Kenya	RO	Romania
CA	Canada	KG	Kyrgyzstan	RU	Russian Federation
CF	Central African Republic	KP	Democratic People's Republic of Korea	SD	Sudan
CG	Congo	KR	Republic of Korea	SE	Sweden
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LI	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LU	Luxembourg	TD	Chad
CS	Czechoslovakia	LV	Latvia	TG	Togo
CZ	Czech Republic	MC	Monaco	TJ	Tajikistan
DE	Germany	MD	Republic of Moldova	TT	Trinidad and Tobago
DK	Denmark	MG	Madagascar	UA	Ukraine
ES	Spain	ML	Mali	US	United States of America
FI	Finland	MN	Mongolia	UZ	Uzbekistan
FR	France			VN	Viet Nam
GA	Gabon				

DATA COMPRESSION AND DECOMPRESSION

CROSS REFERENCE TO APPENDICES

5 Appendix A, which is a part of the present disclosure, is a listing of a software implementation written in the programming language C.

 Appendices B-1 and B-2, which are part of the present disclosure, together are a description of a hardware
10 implementation in the commonly used hardware description language ELLA.

 Appendix C, which is part of the present disclosure is a listing of a software implementation written in the programming language C and assembly code.

15 A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document, but otherwise reserves all copyright rights whatsoever.

20 FIELD OF THE INVENTION

 This invention relates to a method of and apparatus for data compression and decompression. In particular, this invention relates the compression, decompression, transmission and storage of audio, still-image and video
25 data in digital form.

BACKGROUND INFORMATION

 An image such as an image displayed on a computer monitor may be represented as a two-dimensional matrix of digital data values. A single frame on a VGA computer
30 monitor may, for example, be represented as three matrixes of pixel values. Each of the three matrixes has a data value which corresponds to a pixel on the monitor.

 The images on the monitor can be represented by a 640 by 480 matrix of data values representing the luminance

- 2 -

(brightness) values Y of the pixels of the screen and two other 640 by 480 matrixes of data values representing the chrominance (color) values U and V of the pixels on the screen. Although the luminance and chrominance values are 5 analog values, the one luminance value and the two chrominance values for a pixel may be digitized from analog form into discrete digital values. Each luminance and chrominance digital value may be represented by an 8-bit number. One frame of a computer monitor therefore 10 typically requires about 7 megabits of memory to store in an uncompressed form.

In view of the large amount of memory required to store or transmit a single image in uncompressed digital form, it would be desirable to compress the digital image 15 data before storage or transmission in such a way that the compressed digital data could later be decompressed to recover the original image data for viewing. In this way, a smaller amount of compressed digital data could be stored or transmitted. Accordingly, numerous digital 20 image compression and decompression methods have been developed.

According to one method, each individual digital value is converted into a corresponding digital code. Some of the codes have a small number of bits whereas 25 others of the codes have a larger number of bits. In order to take advantage of the fact that some of the codes are short whereas others of the codes are longer, the original digital data values of the original image are filtered using digital filters into a high frequency component and 30 a low frequency component. The high frequency component represents ambiguities in the image and is therefore observed to have a comparatively large number of identical data values for real-world images. By encoding the commonly occurring digital data values in the high 35 frequency component with the short digital codes, the total number of bits required to store the image data can be reduced from the number of bits that would otherwise be

- 3 -

required if 8-bits were used to represent all of the data values. Because the total number of bits in the resulting encoded data is less than the total number of bits in the original sequence of data values, the original image is said to have been compressed.

To decompress the compressed encoded data to recover the original image data, the compressed encoded data is decoded using the same digital code. The resulting high and low frequency components are then recombined to form the two-dimensional matrix of original image data values.

Where the data being compressed is two-dimensional data such as image data, separation of the original data into high and low frequency components by the digital filters may be accomplished by filtering in two dimensions such as the horizontal dimension of the image and the vertical dimension of the image. Similarly, decoded high and low frequency components can be recombined into the original image data values by recombining in two dimensions.

To achieve even greater compression, the low frequency component may itself be filtered into its high and low frequency components before encoding. Similarly, the low frequency component of the low frequency component may also be refiltered. This process of recursive filtering may be repeated a number of times. Whether or not recursive filtering is performed, the filtered image data is said to have been "transformed" into the high and low frequency components. This digital filtering is called a "transform". Similarly, the high and low pass components are said to be "inverse transformed" back into the original data values. This process is known as the "inverse transform".

Figure 1 is a diagram of a digital gray-scale image of a solid black square 1 on a white background 2 represented by a 640 by 480 matrix of 8-bit data luminance values.

Figure 2 is a diagram illustrating a first

- 4 -

intermediate step in the generation of the high and low frequency components of the original image. A high pass digital filter which outputs a single data value using multiple data values as inputs is first run across the original image values from left to right, row by row, to generate G subblock 3. The number of digital values in G subblock 3 is half of the number of data values in the original image of Figure 1 because the digital filter is sequentially moved to the right by twos to process two additional data values for each additional one data output generated for G subblock 3. Similarly, a low pass digital filter which outputs a single data value using multiple data values as inputs is first run across the original image values from left to right, row by row, to generate H subblock 4. The number of digital values in H subblock 4 is half of the number of data values in the original image because the digital filter is moved to the right by twos to process two additional data values for each additional one data output generated for H subblock 4. Each of the two vertical bars in high pass G subblock 3 appears where a change occurs spatially in the horizontal dimension in the original image of Figure 1. Where the G filter encounters a change from white data values to black data values when the filter G is run across the image of Figure 1 in a horizontal direction, the G digital filter outputs a corresponding black data value into subblock 3. Similarly, when the G digital filter encounters the next change, which is this time a change from black to white data values, the G digital filter again outputs a corresponding black data value into G subblock 3.

Figure 3 is a diagram illustrating a second intermediate step in the generation of the high and low frequency components of the original image. The high pass digital filter is run down the various columns of the subblocks H and G of Figure 2 to form the HG subblock 5 and GG subblock 6 shown in Figure 3. Similarly, the low pass digital filter is run down the various columns of the

- 5 -

H and G subblocks 3 and 4 of Figure 2 to form HH and GH subblocks 7 and 8 shown in Figure 3. The result is the low pass component in subblock HH and the three high pass component subblocks GH, HG and GG. The total number of high and low pass component data values in Figure 3 is equal to the number of data values in the original image of Figure 1. The data values in the high pass component subblocks GH, HG and GG are referred to as the high frequency component data values of octave 0.

10 The low pass subblock HH is then filtered horizontally and vertically in the same way into its low and high frequency components. Figure 4 illustrates the resulting subblocks. The data values in HHHG subblock 9, HHGH subblock 10, and HHGG subblock 11 are referred to as
15 the high frequency component data values of octave 1.

Subblock HHHH is the low frequency component. Although not illustrated, the low frequency HHHH subblock 12 can be refiltered using the same method. As can be seen from Figure 4, the high frequency components of octaves 0 and 1
20 are predominantly white because black in these subblocks denotes changes from white to black or black to white in the data blocks from which the high frequency subblocks are generated. The changes, which are sometimes called edges, from white to black as well as black to white in Figure 1
25 result in high frequency data values in the HG, HG and GG quadrants as illustrated in Figure 3.

Once the image data has been filtered the desired number of times using the above method, the resulting transformed data values are encoded using a digital code
30 such as the Huffman code in Table 1.

- 6 -

	<u>Corresponding</u> <u>Gray-Scale</u>	<u>Digital</u> <u>Value</u>	<u>Digital</u> <u>Code</u>
		.	
		.	
5		5	1000001
		4	100001
		3	10001
		2	1001
10	black	1	101
	white	0	0
		-1	111
		-2	1101
		-3	11001
15		-4	110001
		-5	1100001
		.	
		.	
		.	

20

Table 1

Because the high frequency components of the original image of Figure 1 are predominantly white as is evident from Figures 3 and 4, the gray-scale white is assigned the single bit 0 in the above digital code. The next most common gray-scale color in the transformed image is black. Accordingly, gray-scale black is assigned the next shortest code of 101. The image of Figure 1 is comprised only of black and white pixels. If the image were to involve other gray-scale shades, then other codes would be used to encode those gray-scale colors, the more predominant gray-scale shades being assigned the relatively shorter codes. The result of the Huffman encoding is that the digital values which predominate in the high frequency components are coded into codes having a few number of bits. Accordingly, the number of bits required to represent the original image data is reduced. The image is therefore said to have been compressed.

Problems occur during compression, however, when the digital filters operate at the boundaries of the data values. For example, when the high pass digital filter generating the high pass component begins generating high pass data values of octave 0 at the left hand side of the original image data, some of the filter inputs required by

- 7 -

the filter do not exist.

Figure 5 illustrates the four data values required by a four coefficient high pass digital filter G in order to generate the first high pass data value G_0 of octave 0. As shown in Figure 5, data values D_1 , D_2 , D_3 and D_4 are required to generate the second high pass data value of octave 0, data value G_1 . In order to generate the first high pass component output data value G_0 , on the other hand, data values D_{-1} , D_0 , D_1 , and D_2 are required. Data value D_{-1} does not, however, exist in the original image data.

Several techniques have been developed in an attempt to solve the problem of the digital filter extending beyond the boundaries of the image data being transformed. In one technique, called zero padding, the nonexistent data values outside the image are simply assumed to be zeros. This may result in discontinuities at the boundary, however, where an object in the image would otherwise have extended beyond the image boundary but where the assumed zeros cause an abrupt truncation of the object at the boundary. In another technique, called circular convolution, the two dimensional multi-octave transform can be expressed in terms of one dimensional finite convolutions. Circular convolution joins the ends of the data together. This introduces a false discontinuity at the join but the problem of data values extending beyond the image boundaries no longer exists. In another technique, called symmetric circular convolution, the image data at each data boundary is mirrored. A signal such as a ramp, for example, will become a peak when it is mirrored. In another technique, called doubly symmetric circular convolution, the data is not only mirrored spatially but the values are also mirrored about the boundary value. This method attempts to maintain continuity of both the signal and its first derivative but requires more computation for the extra mirror because the mirrored values must be pre-calculated

- 8 -

before convolution.

Figure 6 illustrates yet another technique which has been developed to solve the boundary problem. According to this technique, the high and low pass digital filters 5 are moved through the data values in a snake-like pattern in order to eliminate image boundaries in the image data. After the initial one dimensional convolution, the image contains alternating columns of low and high pass information. By snaking through the low pass sub-band 10 before the high pass, only two discontinuities are introduced. This snaking technique, however, requires reversing the digital filter coefficients on alternate rows as the filter moves through the image data. This changing of filter coefficients as well as the requirement 15 to change the direction of movement of the digital filters through various blocks of data values makes the snaking technique difficult to implement. Accordingly, an easily implemented method for solving the boundary problem is sought which can be used in data compression and 20 decompression.

Not only does the transformation result in problems at the boundaries of the image data, but the transformation itself typically requires a large number of complex computations and/or data rearrangements. The time 25 required to compress and decompress an image of data values can therefore be significant. Moreover, the cost of associated hardware required to perform the involved computations of the forward transform and the inverse transform may be so high that the transform method cannot 30 be used in cost-sensitive applications. A compression and decompression method is therefore sought that not only successfully handles the boundary problems associated with the forward transform and inverse transform but also is efficiently and inexpensively implementable in hardware 35 and/or software. The computational complexity of the method should therefore be low.

In addition to transformation and encoding, even

- 9 -

further compression is possible. A method known as tree encoding may, for example, be employed. Moreover, a method called quantization can be employed to further compress the data. Tree encoding and quantization are
5 described in various texts and articles including "Image Compression using the 2-D Wavelet Transform" by A.S. Lewis and G. Knowles, published in IEEE Transactions on Image Processing, April 1992. Furthermore, video data which comprises sequences of images can be compressed by taking
10 advantage of the similarities between successive images. Where a portion of successive images does not change from one image to the next, the portion of the first image can be used for the next image, thereby reducing the number of bits necessary to represent the sequence of images.

15 JPEG (Joint Photographics Experts Group) is an international standard for still-images which typically achieves about a 10:1 compression ratios for monochrome images and 15:1 compression ratios for color images. The JPEG standard employs a combination of a type of Fourier
20 transform, known as the discrete-cosine transform, in combination with quantization and a Huffman-like code. MPEG1 (Motion Picture Experts Group) and MPEG2 are two international video compression standards. MPEG2 is a standard which is still evolving which is targeted for
25 broadcast television. MPEG2 allows the picture quality to be adjusted to allow more television information to be transmitted, e.g., on a given coaxial cable. H.261 is another video standard based on the discrete-cosine transform. H.261 also varies the amount of compression
30 depending on the data rate required.

Compression standards such as JPEG, MPEG1, MPEG2 and H.261 are optimized to minimize the signal to noise ratio of the error between the original and the reconstructed image. Due to this optimization, these methods are very
35 complex. Chips implementing MPEG1, for example, may be costly and require as many as 1.5 million transistors. These methods only partially take advantage of the fact

- 10 -

that the human visual system is quite insensitive to signal to noise ratio. Accordingly, some of the complexity inherent in these standards is wasted on the human eye. Moreover, because these standards encode by 5 areas of the image, they are not particularly sensitive to edge-type information which is of high importance to the human visual system. In view of these maladaptions of current compression standards to the characteristics of the human visual system, a new compression and 10 decompression method is sought which handles the above-described boundary problem and which takes advantage of the fact that the human visual system is more sensitive to edge information than signal to noise ratio so that the complexity and cost of implementing the method can be 15 reduced.

SUMMARY

A compression and decompression method using wavelet decomposition, frequency based tree encoding, tree based motion encoding, frequency weighted quantization, Huffman 20 encoding, and tree based activity estimation for bit rate control is disclosed. Forward and inverse quasi-perfect reconstruction transforms are used to generate the wavelet decomposition and to reconstruct data values close to the original data values. The forward and inverse quasi- 25 perfect reconstruction transforms utilize special filters at the boundaries of the data being transformed and/or inverse transformed to solve the above-mentioned boundary problem.

In accordance with some embodiments of the present 30 invention, a decompression method uses four coefficient inverse perfect reconstruction digital filters. The coefficients of these inverse perfect reconstruction digital filters require a small number of additions to implement thereby enabling rapid decompression in software 35 executing on a general purpose digital computer having a microprocessor. The method partially inverse transforms a

- 11 -

sub-band decomposition to generate a small low frequency component image. This small image is expanded in one dimension by performing interpolation on the rows of the small image and is expanded in a second dimension by replicating rows of the interpolated small image. Transformed chrominance data values are inverse transformed using inverse perfect reconstruction digital filters having a fewer number of coefficients than the inverse perfect reconstruction digital filters used to inverse transform the corresponding transformed luminance data values. In one embodiment, two coefficient Haar digital filters are used as the inverse perfect reconstruction digital filters which inverse transform transformed chrominance data values. Variable-length tokens are used in the compressed data stream to indicate changes in encoding methods used to encode data values in the compressed data stream.

BRIEF DESCRIPTION OF THE DRAWINGS

Figures 1-4 (Prior Art) are diagrams illustrating a sub-band decomposition of an image.

Figure 5 (Prior Art) is a diagram illustrating a boundary problem associated with the generation of prior art sub-band decompositions.

Figure 6 (Prior Art) is a diagram illustrating a solution to the boundary problem associated with the generation of prior art sub-band decompositions.

Figure 7 is a diagram illustrating a one-dimensional decomposition.

Figures 8 and 9 are diagrams illustrating the separation of an input signal into a high pass component and a low pass component.

Figures 10, 11, 14 and 15 are diagrams illustrating a transformation in accordance with one embodiment of the present invention.

Figures 12 and 13 are diagrams illustrating the operation of high pass and low pass forward transform

- 12 -

digital filters in accordance with one embodiment of the present invention.

Figure 16 is a diagram of a two-dimensional matrix of original data values in accordance with one embodiment of 5 the present invention.

Figure 17 is a diagram of the two-dimensional matrix of Figure 16 after one octave of forward transform in accordance with one embodiment of the present invention.

Figure 18 is a diagram of the two-dimensional matrix 10 of Figure 16 after two octaves of forward transform in accordance with one embodiment of the present invention.

Figures 19 and 20 are diagrams illustrating a boundary problem solved in accordance with one embodiment of the present invention.

15 Figure 21 is a diagram illustrating the operation of boundary forward transform digital filters in accordance with one embodiment of the present invention.

Figure 22 is a diagram illustrating the operation of start and end inverse transform digital filters in 20 accordance with one embodiment of the present invention.

Figure 23 is a diagram illustrating a one-dimensional tree structure in accordance one embodiment of the present invention.

Figure 24A-D are diagrams illustrating the recursive 25 filtering of data values to generate a one-dimensional decomposition corresponding with the one-dimensional tree structure of Figure 23.

Figure 25 is a diagram of a two-dimensional tree structure of two-by-two blocks of data values in 30 accordance with one embodiment of the present invention.

Figure 26 is a pictorial representation of the data values of the two-dimension tree structure of Figure 25.

Figures 27-29 are diagrams illustrating a method and apparatus for determining the addresses of data values of 35 a tree structure in accordance with one embodiment of the present invention.

Figure 30 and 31 are diagrams illustrating a

- 13 -

quantization of transformed data values in accordance with one embodiment of the present invention.

Figures 32 and 33 are diagrams illustrating the sensitivity of the human eye to spatial frequency.

5 Figures 34 is a diagram illustrating the distribution of high pass component data values in a four octave wavelet decomposition of the test image Lenna.

Figure 35 is a diagram illustrating the distribution of data values of the test image Lenna before wavelet
10 transformation.

Figure 36 is a block diagram illustrating a video encoder and a video decoder in accordance with one embodiment of the present invention.

Figure 37 is a diagram illustrating modes of the
15 video encoder and video decoder of Figure 36 and the corresponding token values.

Figure 38 is a diagram illustrating how various flags combine to generate a new mode when the inherited mode is send in accordance with one embodiment of the present
20 invention.

Figures 39-40 are diagrams of a black box on a white background illustrating motion.

Figures 41-43 are one-dimensional tree structures corresponding to the motion of an edge illustrated in
25 Figures 39-40.

Figure 44 is a diagram illustrating variable-length tokens in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

30 QUASI-PERFECT RECONSTRUCTION FILTERS

The wavelet transform was introduced by Jean Morlet in 1984 to overcome problems encountered in analyzing geological signals. See "Cycle-octave and Related Transforms In Seismic Signal Analysis", Goupillaud,
35 Grossman and Morlet, Geoeexploration, vol. 23, 1984. Since then, the wavelet transform has been a new and exciting

- 14 -

method of analyzing signals and has already been applied to a wide range of tasks such as quantum mechanics and signal processing. The wavelet transform has a number of advantages over more traditional Fourier techniques principally used today in the analysis of signals. The wavelet transform and the high and low pass four coefficient quasi-perfect reconstruction filters of the present invention are therefore described by relating them to the windowed Fourier transform.

10 The windowed Fourier transform is the principle transform used today to analyze the spectral components of a signal. The Fourier transform decomposes a signal under analysis into a set of complex sinusoidal basis functions. The resulting Fourier series can be interpreted as the
15 frequency spectra of the signal. The continuous Fourier transform is defined as follows:

$$F(\omega) = \int_{-\infty}^{\infty} e^{-j\omega t} f(t) dt \quad (\text{equ. 1})$$

Where $f(t)$ is the time domain signal under analysis and $F(\omega)$ is the Fourier transform of the signal under
20 analysis. Although many applications require an estimate of the spectral content of an input signal, the above formula is impractical for most systems. In order to calculate the Fourier transform, the input signal $f(t)$ must be defined for all values of time t , whereas in most
25 practical systems, $f(t)$ is only defined over a finite range of time.

Several methods have therefore been devised to transform the finite input signal into an infinite signal so that the Fourier transform can be applied. The
30 windowed Fourier transform is one such solution. The windowed Fourier transform is defined as follows:

$$F_w(\omega, \tau) = \int_{-\infty}^{\infty} w(t-\tau) e^{-j\omega t} f(t) dt \quad (\text{equ. 2})$$

Where $f(t)$ is the time domain signal under analysis,

- 15 -

$F_w(\omega, \tau)$ is the windowed Fourier transform of the time domain signal under analysis, and $w(t)$ is the windowing function. The windowing function is usually chosen to be zero outside an interval of finite length. Alternatively, as the spectral content of the input $f(t)$ varies with time, the input signal can be examined by performing the transform at time τ using a more local window function. In either case, the output transform is the convolution of the window function and the signal under analysis so that the spectra of the window itself is present in the transform results. Consequently, the windowing function is chosen to minimize this effect. Looking at this technique from another viewpoint, the basis functions of a windowed Fourier transform are not complex sinusoids but rather are windowed complex sinusoids. Dennis Gabor used a real Gaussian function in conjunction with sinusoids of varying frequencies to produce a complete set of basis functions (known as Gabor functions) with which to analyze a signal. For a locality given by the effective width of the Gaussian function, the sinusoidal frequency is varied such that the entire spectrum is covered.

The wavelet transform decomposes a signal into a set of basis functions that can be nearly local in both frequency and time. This is achieved by translating and dilating a function $\Psi(t)$ that has spatial and spectral locality to form a set of basis functions:

$$\sqrt{s}\Psi(s(t-u)) \quad (\text{equ. 3})$$

wherein s and u are real numbers and are the variables of the transform. The function $\Psi(t)$ is called the wavelet.

The continuous wavelet transform of a signal under analysis is defined as follows:

$$W(s, u) = \sqrt{s} \int_{-\infty}^{\infty} \Psi(s(t-u)) f(t) dt \quad (\text{equ. 4})$$

Where $f(t)$ is the time domain signal under analysis,

- 16 -

W(s,u) is its wavelet transform, Ψ is the wavelet, s is the positive dilation factor and u is the scaled translation distance. The spatial and spectral locality of the wavelet transform is dependent on the characteristics of the wavelet.

Because the signal under analysis in the compression of digitally sampled images has finite length, the discrete counterpart of the continuous wavelet transform is used. The wavelet transform performs a multiresolution decomposition based on a sequence of resolutions often referred to as "octaves". The frequencies of consecutive octaves vary uniformly on a logarithmic frequency scale. This logarithmic scale can be selected so that consecutive octaves differ by a factor of two in frequency. The basis functions are:

$$\{\psi^j(x-2^{-j}n)\} \text{ for } (j,n) \in \mathbb{Z}^2 \quad (\text{equ. 5})$$

where \mathbb{Z} is the set of all integers, $\mathbb{Z}^2 = \{(j,n) : j,n \in \mathbb{Z}\}$, and $\psi^j(x) = \sqrt{2^j} \psi(2^j x)$.

In a sampled system, a resolution r signifies that the signal under analysis has been sampled at r samples per unit length. A multiresolution analysis studies an input signal at a number of resolutions, which in the case of the present invention is the sequence $r = 2^j$ where $j \in \mathbb{Z}$. The difference in frequency between consecutive octaves therefore varies by a factor of two.

Stephane Mallat formalized the relationship between wavelet transforms and multiresolution analysis by first defining a multiresolution space sequence $\{V_j\}_{j \in \mathbb{Z}}$, where V_j is the set of all possible approximated signals at resolution 2^j . He then showed that an orthonormal basis for V_j can be constructed by $\{\phi^j(x-2^{-j}n)\}_{n \in \mathbb{Z}}$. $\phi(x)$ is called the scaling function where for any $j \in \mathbb{Z}$, $\phi^j(x) = \sqrt{2^j} \phi(2^j x)$. He then showed that a signal $f(x)$ can be approximated at a resolution 2^j by the set of samples:

- 17 -

$$S_j = \{\sqrt{2^j} \langle f, \phi_n^j \rangle\}_{n \in \mathbb{Z}} \quad (\text{equ. 6})$$

where $\langle f, g \rangle = \int_{-\infty}^{\infty} f(x) g(x) dx$, where $f, g \in L^2(\mathbb{R})$,
 the set of square integrable functions on \mathbb{R} . This is
 equivalent to convolving the signal $f(x)$ with the scaling
 5 function $\phi(-x)$ at a sampling rate of 2^j . However, this
 representation is highly redundant because $V_j \subset V_{j+1}, j \in \mathbb{Z}$.
 It would be more efficient to generate a sequence of
 multiresolution detail signals O_j which represents the
 difference information between successive resolutions
 10 $O_j \oplus V_j = V_{j+1}$ where O_j is orthogonal to V_j . Mallat proved that
 there exists a function $\Psi(x)$ called the wavelet where:

$$\Psi^j(x) = \sqrt{2^j} \Psi(2^j x) \quad (\text{equ. 7})$$

such that $\{\Psi(x-2^j n)\}_{n \in \mathbb{Z}}$ is an orthonormal basis of O_j and
 $\{\Psi(x-2^j n)\}, (j, n) \in \mathbb{Z}^2$, is an orthonormal basis of $L^2(\mathbb{R})$.
 15 The detail signal at resolution 2^{j+1} is represented by the
 set of data values:

$$N_j = \{\sqrt{2^j} \langle f, \Psi_n^j \rangle\}_{n \in \mathbb{Z}} \quad (\text{equ. 8})$$

which is equivalent to convolving the signal $f(x)$ with the
 wavelet $\Psi(-x)$ at a sampling rate of 2^j .

20 Hence, the original signal $f(x)$ can be completely
 represented by the sets of data values $(S_j, (N_j)_{J \leq j \leq -1})$,
 where $J < 0$ gives the number of octaves. This
 representation in the form of data values is known as the
 discrete wavelet decomposition. The S_j notation used by
 25 Mallat refers to recursively low pass filter values of the
 original signal. S_0 corresponds to the original data
 values D . S_1 corresponds to the H data values from the
 low pass filter. N_1 corresponds to the G data values from
 the high pass filter. S_j corresponds to the next low pass
 30 filtered values from the previous H sub-band. N_j
 corresponds to the next high pass filtered values from the
 previous H sub-band.

If the sampling patterns of the discrete windowed

- 18 -

Fourier transform and the discrete wavelet transform are compared while maintaining the spatial locality of the highest frequency sample for both transforms, then the efficiency of the discrete wavelet decomposition is revealed. The window Fourier transform produces a linear sampling grid, each data value being a constant spatial distance or a constant frequency away from its neighbor. The result is a heavy over-sampling of the lower frequencies. The wavelet transform, in contrast, samples each of its octave wide frequency bands at the minimum rate such that no redundant information is introduced into the discrete wavelet decomposition. The wavelet transform is able to achieve highly local spatial sampling at high frequencies by the use of octave wide frequency bands. At low frequencies, spectral locality takes precedence over spatial locality.

Figure 7 illustrates the spatial and spectral locality of a sequence of sampled data values. The box surrounding a data value represents the spatial and spectral locality of the data value. The regions of Figure 7 are presented for explanation purposes. In reality there is some overlap and aliasing between adjacent data values, the characteristics of which are determined by the particular wavelet function used.

Mallat showed the wavelet transform can be computed with a pyramid technique, where only two filters are used. Using this technique, S_j and N_j are calculated from S_{j+1} , S_j being used as the input for the next octave of decomposition. A low pass filter H :

$$h(n) = \frac{1}{\sqrt{2}} \langle \phi_0^{-1}, \phi_n^0 \rangle \quad (\text{equ. 9})$$

Mallat showed that S_j can be calculated by convolving from S_{j+1} with H and keeping every other output (i.e. sub-sampling by a factor of 2).

A method for calculating N_j from S_{j+1} can also be derived. This method involves convolving S_{j+1} with a high

- 19 -

pass filter G and sub-sampling by a factor of 2. The high pass filter G is defined by the following coefficients:

$$g(n) = (-1)^{1-n} h(1-n) \quad (\text{equ. 10})$$

The relationship between the H and G filters results in a large saving when the filters are implemented in hardware.

Figures 8 and 9 illustrate that these two filters H and G form a complementary pair that split an input signal into two half band output signals. Both the high and the low pass outputs can be sub-sampled by a factor of two without corrupting the high frequency information because any aliasing introduced by the sub-sampling will be corrected in the reconstruction. There are the same number of filtered data values as there are original image data values.

The particular wavelet which is best in analyzing a signal under analysis is heavily dependent on the characteristics of the signal under analysis. The closer the wavelet resembles the features of the signal, the more efficient the wavelet representation of the signal will be. In addition, reconstruction errors introduced by quantization resemble the wavelet. Typically, the amount of aliasing varies with spatial support (the number of coefficients of the wavelet filters). Long wavelets can be constructed such that aliasing between adjacent octave bands is minimized. However, the spatial equivalent of aliasing, overlap, increases with filter length. Conversely, short wavelets have little or no overlap spatially but exhibit large amounts of aliasing in the frequency domain. To properly determine the suitability of a wavelet for a particular application, these factors of size and shape must be considered.

To apply the wavelet transform to image processing, the present invention employs a particular wavelet called the four coefficient Daubechies wavelet. Because the four

- 20 -

coefficient Daubechies wavelet has only four coefficients, it is very short. This is well-suited for analyzing important image features such as object edges. Edges by definition are spatially local discontinuities. Edges often consist of a wide spectral range which, when filtered through a high pass filter, give rise to relatively larger filtered outputs only when the analysis filter coincides with the edge. When the analysis filter does not coincide with the edge, relatively smaller filtered outputs are output by the filter. The shorter the analysis filter used, the more finely the spatial position of the edge is resolved. Longer filters produce more of the relatively larger data values to represent an edge. The shortness of the filter also makes the transform calculation relatively inexpensive to implement compared with that of longer filters or image transformations such as the Fourier or discrete cosine transforms. The four coefficient Daubechies wavelet was selected for use only after a careful analysis of both its spatial and aliasing characteristics. Longer wavelets such as the six coefficient Daubechies wavelet could, however, also be used if a more complex implementation were acceptable. Short filters such as the two coefficients Haar wavelet could also be used if the attendant high levels of noise were acceptable.

The true coefficients of the four coefficient Daubechies wavelet are:

$$a = \frac{1+\sqrt{3}}{8}, b = \frac{3+\sqrt{3}}{8}, c = \frac{3-\sqrt{3}}{8}, d = \frac{-1+\sqrt{3}}{8} \quad (\text{equ. 11})$$

The low pass four coefficient Daubechies digital filter is given by:

$$H\left(\frac{x}{2}\right) = aD(x-1) + bD(x) + cD(x+1) - dD(x+2) \quad (\text{equ. 12})$$

The high pass four coefficient Daubechies digital filter is given by:

- 21 -

$$G\left(\frac{x}{2}\right) = dD(x-1) + cD(x) - bD(x+1) + aD(x+2) \quad (\text{equ. 13})$$

In equations 12 and 13, $D(x-1)$, $D(x)$, $D(x+1)$ and $D(x+2)$ are four consecutive data values. $H\left(\frac{x}{2}\right)$ and $G\left(\frac{x}{2}\right)$ are true perfect reconstruction filters, i.e. the inverse transform
 5 perfectly reconstructs the original data. For example, when the filters operate on data values $D(1)$, $D(2)$, $D(3)$ and $D(4)$, outputs $H(1)$ and $G(1)$ are generated. Index x in this case would be 2. Due to the presence of the $\frac{x}{2}$ as the index for the filters H and G , the values of x can
 10 only be even integers.

To simplify the computational complexity involved in performing the transformation on real data, the coefficients of the four coefficient Daubechies filter which are non-rational numbers are converted into rational
 15 numbers which can be efficiently implemented in software or hardware. Floating point coefficients are not used because performing floating point arithmetic is time consuming and expensive when implemented in software or hardware.

20 To convert the four Daubechies coefficients for implementation, three relationships of the coefficients a , b , c and d are important. In order for the H filter to have unity gain, the following equation must hold:

$$a + b + c - d = 1 \quad (\text{equ. 14})$$

25 In order for the G filter to reject all zero frequency components in the input data values, the following equation must hold:

$$a - b + c + d = 0 \quad (\text{equ. 15})$$

In order for the resulting H and G filters to be able to
 30 generate a decomposition which is perfectly reconstructible into the original image data the following equation must hold:

- 22 -

$$ac - bd = 0$$

(equ. 16)

True four coefficient Daubechies filters satisfy the above three equations 14, 15, and 16. However, when the coefficients of the true low and high pass four
 5 coefficient Daubechies filters are converted for implementation, at least one of the three relationships must be broken. In the preferred embodiment, unity gain and the rejection of all zero frequency components are maintained. It is the third relationship of equation 16
 10 that is compromised. Perfect reconstruction is compromised because the process of compressing image data itself inherently introduces some noise due to the tree coding and quantization of the present invention. The reconstructed data values therefore necessarily involve
 15 noise when a real-world image is compressed and then reconstructed. We define filters which satisfy equations 14, and 15 and approximately satisfy equation 16, quasi-perfect reconstruction filters.

Table 2 illustrates a process of converting the
 20 coefficients a, b, c and d for implementation.

$$a = \frac{1+\sqrt{3}}{8} = .3415(32) = 10.92 = \frac{11}{32}$$

$$b = \frac{3+\sqrt{3}}{8} = .5915(32) = 18.92 = \frac{19}{32}$$

$$c = \frac{3-\sqrt{3}}{8} = .1585(32) = 5.072 = \frac{5}{32}$$

$$25 \quad d = \frac{-1+\sqrt{3}}{8} = .0915(32) = 2.928 = \frac{3}{32}$$

Table 2

The true four coefficient Daubechies filter coefficients are listed in the left hand column of Table 2. In the next column to the right, the true coefficients are shown
 30 rounded to four places beyond the decimal point. The

- 23 -

rounded coefficients are scaled by a factor of 32 to achieve the values in the next column to the right. From each value in the third column, an integer value is selected. Which integers are selected has a dramatic effect on the complexity of the software or hardware which compresses the image data. The selected integers are divided by 32 so that the scaling by 32 shown in the second column does not change the values of the resulting converted coefficients.

10 In selecting the integers for the fourth column, the relationship of the three equations 14, 15 and 16 are observed. In the case of $a = 11/32$, $b = 19/32$, $c = 5/32$ and $d = 3/32$, the relationships $a+b+c-d=1$ and $a-b+c+d=0$ both are maintained. Because the converted coefficients in the rightmost column of Table 2 are quite close to the true coefficient values in the leftmost column, the resulting four coefficient filters based on coefficients a , b , c and d allow near perfect reconstruction. On a typical 640 by 480 image, the error between the original and reconstructed data values after forward and then inverse transformation has been experimentally verified to exceed 50 dB.

The resulting high pass four coefficient quasi-Daubechies filter is:

$$25 \quad H\left(\frac{x}{2}\right) = \frac{11}{32}D(x-1) + \frac{19}{32}D(x) + \frac{5}{32}D(x+1) - \frac{3}{32}D(x+2) \text{ (equ. 17)}$$

The resulting low pass four coefficient quasi-Daubechies filter is:

$$G\left(\frac{x}{2}\right) = \frac{3}{32}D(x-1) + \frac{5}{32}D(x) - \frac{19}{32}D(x+1) + \frac{11}{32}D(x+2) \text{ (equ. 18)}$$

Because the high and low pass four coefficient quasi-Daubechies filters satisfy equations 14 and 15 and approximately satisfy equation 16, the high and low pass four coefficient quasi-Daubechies filters are quasi-perfect reconstruction filters.

- 24 -

Note that the particular converted coefficients of the quasi-Daubechies filters of equations 17 and 18 result in significant computational simplicity when implementation is either software and/or hardware.

5 Multiplications and divisions by factors of two such as multiplications and divisions by 32 are relatively simple to perform. In either hardware or software, a multiplication by 2 or a division by 2 can be realized by a shift. Because the data values being operated on by the
10 digital filter already exist in storage when the filter is implemented in a typical system, the shifting of this data after the data has been read from storage requires little additional computational overhead. Similarly, changing the sign of a quantity involves little additional
15 overhead. In contrast, multiplication and division by numbers that are not a power of 2 require significant overhead to implement in both software and hardware. The selection of the coefficients in equations 17 and 18 allows $H(x)$ and $G(x)$ to be calculated with only additions
20 and shifts. In other words, all multiplications and divisions are performed without multiplying or dividing by a number which is not a power of 2. Due to the digital filter sequencing through the data values, pipelining techniques can also be employed to reduce the number of
25 adds further by using the sums or differences computed when the filters were operating on prior data values.

Moreover, the magnitudes of the inverse transform filter coefficients are the same as those of the transform filter itself. As described further below, only the order
30 and signs of the coefficients are changed. This reduces the effective number of multiplications which must be performed by a factor of two when the same hardware or software implementation is to be used for both the forward and inverse transform. The fact that the signal being
35 analyzed is being sub-sampled reduces the number of additions by a factor of two because summations are required only on the reading of every other sample. The

- 25 -

effective number of filters is therefore only one to both transform the data into the decomposition and to inverse transform the decomposition back into the image data.

IMAGE COMPRESSION AND DECOMPRESSION USING THE 5 QUASI-PERFECT RECONSTRUCTION TRANSFORM

Color images can be decomposed by treating each Red-Green-Blue (or more usually each Luminance-Chrominance-Chrominance channel) as a separate image. In the case of Luminance-Chrominance-Chrominance (YUV or YIQ) images the
10 chrominance components may already have been sub-sampled. It may be desirable therefore, to transform the chrominance channels through a different number of octaves than the luminance channel. The eye is less sensitive to chrominance at high spatial frequency and therefore these
15 channels can be sub-sampled without loss of perceived quality in the output image. Typically these chrominance channels are sub-sampled by a factor of two in each dimension so that they together take only 50 percent of the bandwidth of the luminance channel. When implementing
20 an image compression technique, the chrominance channels are usually treated the same way as the luminance channel. The compression technique is applied to the three channels independently. This approach is reasonable except in the special cases where very high compression ratios and very
25 high quality output are required. To squeeze the last remaining bits from a compression technique or to achieve more exacting quality criteria, knowledge of how the chrominance rather than luminance values are perceived by the human visual system can be applied to improve the
30 performance of the compression technique by better matching it with the human visual system.

Figure 10 is an illustration of a two dimensional matrix of data values. There are rows of data values extending in the horizontal dimension and there are
35 columns of data values extending in the vertical dimension. Each of the data values may, for example, be

- 26 -

an 8-bit binary number of image pixel information such as the luminance value of a pixel. The data values of Figure 10 represent an image of a black box 100 on a white background 101.

- 5 To transform the data values of the image of Figure 10 in accordance with one aspect of the present invention, a high pass four coefficient quasi-Daubechies digital filter is run across the data values horizontally, row by row, to result in a block 102 of high pass output values G
- 10 shown in Figure 11. The width of the block 102 of high pass output values in Figure 11 is half the width of the original matrix of data values in Figure 10 because the high pass four coefficient quasi-Daubechies digital filter is moved across the rows of the data values by twos.
- 15 Because only one additional digital filter output is generated for each additional two data values processed by the digital filter, the data values of Figure 10 are said to have been sub-sampled by a factor of two.

Figure 12 illustrates the sub-sampling performed by

20 the high pass digital filter. High pass output G_1 is generated by the high pass digital filter from data values D_1 , D_2 , D_3 and D_4 . The next high pass output generated, output G_2 , is generated by the high pass digital filter from data values D_3 , D_4 , D_5 and D_6 . The high pass digital

25 filter therefore moves two data values to the right for each additional high pass output generated.

A low pass four coefficient quasi-Daubechies digital filter is also run across the data values horizontally, row by row, to generate H block 103 of the low pass

30 outputs shown in Figure 11. This block 103 is generated by sub-sampling the data values of Figure 10 in the same way the block 102 was generated. The H and G notation for the low and high pass filter outputs respectively is used as opposed to the S_j and O_j notation used by Mallat to

35 simplify the description of the two-dimensional wavelet transform.

Figure 13 illustrates the sub-sampling of the low

- 27 -

pass digital filter. Low pass output H_1 is generated by the low pass digital filter from data values D_1 , D_2 , D_3 and D_4 . The next low pass output generated, output H_2 , is generated by the low pass digital filter from data values D_3 , D_4 , D_5 and D_6 . The low pass digital filter therefore moves two data values to the right for each additional low pass output generated.

After the high and low pass four coefficient quasi-Daubechies digital filters have generated blocks 102 and 103, the high and low pass four coefficient quasi-Daubechies digital filters are run down the columns of blocks 102 and 103. The values in blocks 102 and 103 are therefore sub-sampled again. The high pass four coefficient quasi-Daubechies digital filter generates blocks 104 and 105. The low pass four coefficient quasi-Daubechies digital filter generates blocks 106 and 107. The resulting four blocks 104-107 are shown in Figure 14. Block 106 is the low frequency component of the original image data. Blocks 107, 104 and 105 comprise the high frequency component of the original image data. Block 106 is denoted block HH. Block 107 is denoted block GH. Block 104 is denoted block HG. Block 105 is denoted block GG.

This process of running the high and low pass four coefficient quasi-Daubechies digital filters across data values both horizontally and vertically to decompose data values into high and low frequency components is then repeated using the data values of the HH block 106 as input data values. The result is shown in Figure 15. Block 108 is the low frequency component and is denoted block HHHH. Blocks 109, 110 and 111 comprise octave 1 of the high frequency component and are denoted HHHG, HHGH, HHGG, respectively. Blocks HG, GH and GG comprise octave 0 of the high frequency component.

Although this recursive decomposition process is only repeated twice to produce high pass component octaves 0 and 1 in the example illustrated in connection with

- 28 -

Figures 10-15, other numbers of recursive decomposition steps are possible. Recursively decomposing the original data values into octaves 0, 1, 2 and 3 has been found to result in satisfactory results for most still image data 5 and recursively decomposing the original data into octaves 0, 1, and 2 has been found to result in satisfactory results for most video image data.

Moreover, the horizontal and subsequent vertical operation of the high and low pass filters can also be 10 reversed. The horizontal and subsequent vertical sequence is explained in connection with this example merely for instructional purposes. The filters can be moved in the vertical direction and then in the horizontal direction. Alternatively, other sequences and dimensions of moving 15 the digital filters through the data values to be processed is possible.

It is also to be understood that if the original image data values are initially arrayed in a two dimensional block as shown in Figure 10, then the 20 processing of the original image data values by the high and low pass filters would not necessarily result in the HH values being located all in an upper right hand quadrant as is shown in Figure 14. To the contrary, depending on where the generated HH values are written, 25 the HH data values can be spread throughout a block. The locations of the HH values are, however, determinable. The HH values are merely illustrated in Figure 14 as being located all in the upper lefthand quadrant for ease of illustration and explanation.

30 Figure 16 is an illustration showing one possible twelve-by-twelve organization of original image data values in a two dimensional array. Figure 16 corresponds with Figure 10. The location in the array of each data value is determined by a row number and column number. A 35 row number and column number of a data value may, for example, correspond with a row address and column address in an addressed storage medium. This addressed storage

- 29 -

medium may, for example, be a semiconductor memory, a magnetic storage medium, or an optical storage medium. The row and column may, for example, also correspond with a pixel location including a location of a pixel on a cathode-ray tube or on a flat panel display.

Figure 17 is an illustration showing the state of the two dimensional array after a one octave decomposition. The HH low frequency components are dispersed throughout the two dimensional array as are the HG values, the GH values, and the GG values. The subscripts attached to the various data values in Figure 17 denote the row and column location of the particular data value as represented in the arrangement illustrated in Figure 14. HH_{00} , HH_{01} , HH_{02} , HH_{03} , HH_{04} and HH_{05} , for example, are six data values which correspond with the top row of data values in HH block 106 of Figure 14. HH_{00} , HH_{10} , HH_{20} , HH_{30} , HH_{40} and HH_{50} , for example, are six data values which correspond with the leftmost column of data values in HH block 106 of Figure 14.

When the high and the low pass forward transform digital filters operate on the four data values D_{01} , D_{02} , D_{03} and D_{04} of Figure 16, the output of the low pass forward transform digital filter is written to location row 0 column 2 and the output of the high pass forward transform digital filter is written to location row 0 column 3. Next, the high and low pass forward transform digital filters are moved two locations to the right to operate on the data values D_{03} , D_{04} , D_{05} and D_{06} . The outputs of the low and high pass forward transform digital filters are written to locations row 0 column 4 and row 0 column 5, respectively. Accordingly, the outputs of the low and high frequency forward transform digital filters are output from the filters to form an interleaved sequence of low and high frequency component data values which overwrite the rows of data values in the two dimensional array.

Similarly, when the low and high pass forward

- 30 -

transform digital filters operate on the four data values at locations column 0, rows 1 through 4, the output of the low pass forward transform digital filter is written to location column 0 row 2. The output of the high pass forward transform digital filter is written to location column 0 row 3. Next the low and high pass forward transform digital filters are moved two locations downward to operate on the data values at locations column 0, rows 3 through 6. The outputs of the low and high pass forward transform digital filters are written to locations column 0 row 4 and column 0 row 5, respectively. Again, the outputs of the low and high pass forward transform digital filters are output from the filters in an interleaved fashion to overwrite the columns of the two dimensional array.

Figure 18 is an illustration showing the state of the two dimensional array after a second octave decomposition. The HHHH low frequency components corresponding which block 108 of Figure 15 as well as the octave 1 high frequency components HHGH, HHHG and HHGG are dispersed throughout the two dimensional array. When the HH values HH_{01} , HH_{02} , HH_{03} and HH_{04} of Figure 17 are processed by the low and high pass forward transform digital filters, the outputs are written to locations row 0 column 4 and row 0 column 6, respectively. Similarly, when the values at locations column 0, rows 2, 4, 6 and 8 are processed by the low and high pass forward transform digital filters, the results are written to locations column 0 row 4 and column 0 row 6, respectively. The data values in Figure 18 are referred to as transformed data values. The transformed data values are said to comprise the decomposition of the original image values.

This method of reading data values, transforming the data values, and writing back the output of the filters is easily expanded to a two dimensional array of a very large size. Only a relatively small number of locations is shown in the two dimensional array of Figures 10-18 for

- 31 -

ease of explanation and clarity of illustration.

The transformed data values are reconverted back into image data values substantially equal to the original image data by carrying out a reverse process. This reverse process is called the inverse transform. Due to the interleaved nature of the decomposition data in Figure 18, the two digital filters used to perform the inverse transform are called interleaved inverse transform digital filters. Odd data values are determined by an odd interleaved inverse digital filter O. Even data values are determined by the even interleaved inverse transform digital filter E.

The odd and even interleaved inverse digital filters can be determined from the low and high pass forward transform digital filters used in the forward transform because the coefficients of the odd interleaved inverse transform digital filters are related to the coefficients of the low and high pass forward transform filters. To determine the coefficients of the odd and even interleaved inverse transform digital filters, the coefficients of the low and high pass forward transform digital filters are reversed. Where the first, second, third and fourth coefficients of the low pass forward transform digital filter H of equation 17 are denoted a, b, c and -d, the first, second, third and fourth coefficients of a reversed filter H* are denoted -d, c, b and a. Similarly, where the first, second, third and fourth coefficients of the high pass forward transform digital filter G of equation 18 are denoted d, c, -b and a, the first, second, third and fourth coefficients of a reverse filter G* are denoted a, -b, c and d.

The first through the fourth coefficients of the even interleaved inverse transform digital filter E are the first coefficient of H*, the first coefficient of G*, the third coefficient of H*, and the third coefficient of G*. The coefficients of the even interleaved inverse transform digital filter E therefore are -d, a, b and c. In the

- 32 -

case of the low and high pass four coefficient quasi-Daubechies filters used in the transform where $a = \frac{11}{32}$, $b = \frac{19}{32}$, $c = \frac{5}{32}$ and $d = \frac{3}{32}$, the even interleaved inverse transform digital filter is:

$$5 \quad \frac{D(2x)}{2} = -\frac{3}{32}H(x-1) + \frac{11}{32}G(x-1) + \frac{19}{32}H(x) + \frac{5}{32}G(x) \text{ (equ. 19)}$$

where $H(x-1)$, $G(x-1)$, $H(x)$ and $G(x)$ are transformed data values of a decomposition to be inverse transformed.

The first through the fourth coefficients of the odd interleaved inverse transform digital filter 0 are the
 10 second coefficient of H^* , the second coefficient of G^* , the fourth coefficient of H^* , and the fourth coefficient of G^* . The coefficients of the odd interleaved inverse transform digital filter 0 therefore are c , $-b$, a and d . In the case of the low and high pass four coefficient
 15 quasi-Daubechies filters used in the transform where $a = \frac{11}{32}$, $b = \frac{19}{32}$, $c = \frac{5}{32}$ and $d = \frac{3}{32}$, the odd interleaved inverse transform digital filter is:

$$\frac{D(2x-1)}{2} = \frac{5}{32}H(x-1) - \frac{19}{32}G(x-1) + \frac{11}{32}H(x) + \frac{3}{32}G(x) \text{ (equ. 20)}$$

where $H(x-1)$, $G(x-1)$, $H(x)$ and $G(x)$ are data values of a
 20 decomposition to be inverse transformed.

To inverse transform the transformed data values of Figure 18 into the data values of Figure 17, the HHHG, HHGG, HHGH and data values are inverse transformed with the HHHH data values to create the HH data values of
 25 Figure 17. This process corresponds with the inverse transformation of HHHG block 109, HHGH block 110, HHGG block 111, and HHHH block 108 of Figure 15 back into the HH data values of block 106 of Figure 14. The HG, GH and GG data values of Figure 18 are therefore not processed by
 30 the odd and even interleaved inverse transform digital filters in this step of the inverse transform.

- 33 -

In Figure 18, the odd interleaved inverse transform digital filter processes the values in locations column 0, rows 0, 2, 4 and 6 to generate the odd data value at location column 0 row 2. The even interleaved inverse transform digital filter data also processes the values in the same locations to generate the even data value at location column 0 row 4. The odd and even interleaved inverse transform digital filters then process the values in locations column 0, rows 4, 6, 8 and A to generate the values at locations column 0 row 6 and column 0 row 8, respectively. Each of the six columns 0, 2, 6, 4, 8, and A of the values of Figure 18 are processed by the odd and even interleaved inverse transform digital filters in accordance with this process.

15 The various locations are then processed again by the odd and even interleaved inverse transform digital filters, this time in the horizontal direction. The odd and even interleaved inverse transform digital filters process the values at locations row 0 columns 0, 2, 4 and 6 to generate the values at locations row 0 column 2 and row 0 column 4, respectively. The odd and even interleaved inverse transform digital filters process the values at locations row 0 columns 4, 6, 8 and A to generate the values at locations row 0 column 6 and row 0 column 8, respectively. Each of the six rows 0, 2, 4 and 8 and of values are processed by the even and odd interleaved inverse transform digital filters in accordance with this process. The result is the reconstruction shown in Figure 17.

30 The even and odd interleaved inverse transform digital filters then process the values shown in Figure 17 into the data values shown in Figure 16. This inverse transformation corresponds with the transformation of the HH block 106, the HG block 104, the GH block 107 and the GG block 105 of Figure 14 into the single block of data value of Figure 10. The resulting reconstructed data values of Figure 16 are substantially equal to the original image

- 34 -

data values.

Note, however, that in the forward transform of the data values of Figure 16 into the data values of Figure 17 that the low and high pass four coefficient quasi-Daubechies digital filters cannot generate all the data values of Figure 17 due to the digital filters requiring data values which are not in the twelve by twelve matrix of data values of Figure 16. These additional data values are said to be beyond the "boundary" of the data values to be transformed.

Figure 19 illustrates the high pass four coefficient quasi-Daubechies digital filter operating over the boundary to generate the G_0 data value. In order to generate the G_0 data value in the same fashion that the other high frequency G data values are generated, the high pass digital filter would require data values D_{-1} , D_0 , D_1 and D_2 as inputs. Data value D_{-1} , however, does not exist. Similarly, Figure 20 illustrates the low pass four coefficient quasi-Daubechies digital filter operating over the boundary to generate the H_0 data value. In order to generate the H_0 data value in the same fashion that the other low frequency H data values are generated, the low pass digital filter would require data values D_{-1} , D_0 , D_1 and D_2 as inputs. Data value D_{-1} , however, does not exist.

The present invention solves this boundary problem by using additional quasi-Daubechies digital filters to generate the data values adjacent the boundary that would otherwise require the use of data values outside the boundary. There is a high pass "start" quasi-Daubechies forward transform digital filter G_s , which is used to generate the first high pass output G_0 . There is a low pass "start" quasi-Daubechies forward transform digital filter H_s , which is used to generate the first low pass output H_0 . These start quasi-Daubechies forward transform digital filters are three coefficient filters rather than four coefficient filters and therefore require only three data values in order to generate an output. This allows

- 35 -

the start quasi-Daubechies forward transform digital filters to operate at the boundary and to generate the first forward transform data values without extending over the boundary.

5 Figure 21 illustrates the low and high pass start quasi-Daubechies forward transform digital filters operating at the starting boundary of image data values D_0 through D_3 . The three coefficient low and high pass start quasi-Daubechies forward transform digital filters operate
10 on data values D_0 , D_1 and D_2 to generate outputs H_0 and G_0 , respectively. H_1 , H_2 , H_3 and H_4 , on the other hand, are generated by the low pass four coefficient quasi-Daubechies forward transform digital filter and G_1 , G_2 , G_3 and G_4 are generated by the high pass four coefficient
15 quasi-Daubechies forward transform digital filter.

A similar boundary problem is encountered at the end of the data values such as at the end of the data values of a row or a column of a two-dimensional array. If the low and high pass four coefficient quasi-Daubechies
20 filters G and H are used at the boundary in the same fashion that they are in the middle of the data values, then the four coefficient quasi-Daubechies forward transform digital filters would have to extend over the end boundary to generate the last low and high pass
25 outputs, respectively.

The present invention solves this boundary problem by using additional quasi-Daubechies forward transform digital filters in order to generate the transformed data values adjacent the end boundary that would otherwise
30 require the use of data outside the boundary. There is a low pass "end" quasi-Daubechies forward transform digital filter H_5 which is used to generate the last low pass output. There is a high pass "end" quasi-Daubechies forward transform digital filter G_5 which is used to
35 generate the last high pass output. These two end quasi-Daubechies forward transform digital filters are three coefficient filters rather than four coefficient filters

- 36 -

and therefore require only three data values in order to generate an output. This allows the end quasi-Daubechies forward transform digital filters to operate at the boundary and to generate the last transform data values 5 without extending over the boundary.

Figure 21 illustrates two low and high pass end quasi-Daubechies forward transform digital filters operating at the end boundary of the image data. These three coefficient low and high pass end quasi-Daubechies 10 forward transform digital filters operate on data values D_0 , D_1 and D_2 to generate outputs H_0 and G_0 , respectively. This process of using the appropriate start or end low or high pass filter is used in performing the transformation at the beginning and at the end of each row and column of 15 the data values to be transformed.

The form of the low pass start quasi-Daubechies forward transform digital filter H_0 is determined by selecting a value of a hypothetical data value D_1 which would be outside the boundary and then determining the 20 value of the four coefficient low pass quasi-Daubechies forward transform filter if that four coefficient forward transform filter were to extend beyond the boundary to the hypothetical data value in such a way as would be necessary to generate the first low pass output H_0 . This 25 hypothetical data value D_1 outside the boundary can be chosen to have one of multiple different values. In some embodiments, the hypothetical data value D_1 has a value equal to the data value D_0 at the boundary. In some embodiments, the hypothetical data value D_1 is set to zero 30 regardless of the data value D_0 . The three coefficient low pass start quasi-Daubechies forward transform digital filter H_0 therefore has the form:

$$H_0 = K1 + bD_0 + cD_1 - dD_2 \quad (\text{equ. 21})$$

where $K1$ is equal to the product aD_0 , where D_0 is the first 35 data value at the start boundary at the start of a

- 37 -

sequence of data values, and where a , b , c and d are the four coefficients of the four coefficient low pass quasi-Daubechies forward transform digital filter. If, for example, hypothetical data value D_1 is chosen to be equal 5 to the data value D_0 adjacent but within the boundary, then $K1=aD_0$ where $a = 11/32$ and D_0 is the data value adjacent the boundary, equation 21 then becomes:

$$H_0 = (a+b)D_0 + cD_1 - dD_2 \quad (\text{equ. 22})$$

The form of the high pass start quasi-Daubechies 10 forward transform digital filter G_1 is determined by the same process using the same hypothetical data value D_1 . The high pass start quasi-Daubechies forward transform digital filter G_1 therefore has the form:

$$G_0 = K2 + cD_0 - bD_1 + aD_2 \quad (\text{equ. 23})$$

15 where $K2$ is equal to the product dD_1 , where D_0 is the first data value at the boundary at the start of a sequence of data values, and where a , b , c and d are the four coefficients of the four coefficient high pass quasi-Daubechies forward transform digital filter. If 20 hypothetical data value D_1 is chosen to be equal to D_0 , then equation 23 becomes:

$$G_0 = (d + c)D_0 - bD_1 + aD_2 \quad (\text{equ. 24})$$

The form of the low pass end quasi-Daubechies forward transform digital filter H_1 is determined in a similar way 25 to the way the low pass start quasi-Daubechies forward transform digital filter is determined. A value of a data value D_c is selected which would be outside the boundary. The value of the four coefficient low pass quasi-Daubechies forward transform digital filter is then 30 determined as if that four coefficient filter were to extend beyond the boundary to data value D_c in such a way

- 38 -

as to generate the last low pass output H_j . The three coefficient low pass end quasi-Daubechies forward transform digital filter therefore has the form:

$$H_j = aD_j + bD_A + cD_B - K3 \quad (\text{equ. 25})$$

5 where $K3$ is equal to the product dD_C , where D_B is the last data value of a sequence of data values to be transformed, and where a , b , c and d are the four coefficients of the four coefficient low pass quasi-Daubechies filter. D_B is the last data value in the particular sequence of data
10 values of this example and is adjacent the end boundary. In the case where the hypothetical data value D_C is chosen to be equal to the data value D_B adjacent but within the end boundary, then $K3=dD_B$ and equation 25 becomes:

$$H_j = aD_j + bD_A + (c-d)D_B \quad (\text{equ. 26})$$

15 The form of the high pass end quasi-Daubechies forward transform digital filter G_j is determined by the same process using the same data value D_C . The three coefficient high pass end quasi-Daubechies forward transform digital filter therefore has the form:

$$20 \quad G_j = dD_j + cD_A - bD_B + K4 \quad (\text{equ. 27})$$

where $K4$ is equal to the product aD_C , where D_B is the last data value in this particular sequence of data values to be transformed, and where a , b , c and d are the four coefficients of the four coefficient high pass quasi-
25 Daubechies forward transform digital filter. D_B is adjacent the end boundary. If hypothetical data value D_C is chosen to be equal to D_B , then equation 27 becomes:

$$G_j = dD_j + cD_A + (-b+a)D_B \quad (\text{equ. 28})$$

It is to be understood that the specific low and high

- 39 -

pass end quasi-Daubechies forward transform digital filters are given above for the case of data values D_0 through D_3 of Figure 21 and are presented merely to illustrate one way in which the start and end digital filters may be determined. In the event quasi-Daubechies filters are not used for the low and high pass forward transform digital filters, the same process of selecting a hypothetical data value or values outside the boundary and then determining the value of a filter as if the filter were to extend beyond the boundary can be used. In some embodiments, multiple hypothetical data values may be selected which would all be required by the digital filters operating on the inside area of the data values in order to produce an output at the boundary. This boundary technique is therefore extendable to various types of digital filters and to digital filters having numbers of coefficients other than four.

As revealed by Figure 22, not only does the forward transformation of data values at the boundary involve a boundary problem, but the inverse transformation of the transformed data values back into original image data values also involves a boundary problem. In the present example where four coefficient quasi-Daubechies filters are used to forward transform non-boundary data values, the inverse transform involves an odd inverse transform digital filter as well as an even inverse transform digital filter. Each of the odd and even filters has four coefficients. The even and odd reconstruction filters alternately generate a sequence of inverse transformed data values.

In Figure 22, the data values to be transformed are denoted $H_0, G_0 \dots H_4, G_4, H_5, G_5$. Where the forward transform processes the rows first and then the columns, the inverse transform processes the columns first and then the rows. Figure 22 therefore shows a column of transferred data values being processed in a first step of the inverse transform. Both the forward and the inverse

- 40 -

transforms in the described example, however, process the columns in a downward direction and process the rows in a left-right direction.

In Figure 22, the inverse transformed data values reconstructed by the inverse transform digital filters are denoted $D_0, D_1, D_2, D_3 \dots D_B$. The odd inverse transform digital filter outputs are shown on the left and the even inverse transform digital filter outputs are shown on the right.

At the beginning of the sequence of data values $H_0, G_0, H_1, G_1 \dots H_i$ and G_i to be inverse transformed, the four coefficient odd and even inverse transform digital filters determine the values of reconstructed data values D_1 and D_2 using values H_0, G_0, H_1 and G_1 , respectively. Reconstructed data value D_0 , however, cannot be reconstructed from the four coefficient even inverse transform digital filter without the four coefficient even inverse transform digital filter extending beyond the boundary. If the four coefficient even inverse transform filter were to be shifted two data values upward so that it could generate data value D_0 , then the even four coefficient inverse transform digital filter would require two additional data values to be transformed, data values G_{-1} and H_{-1} . H_0 is, however, the first data value within the boundary and is located adjacent the boundary.

To avoid the even four coefficient inverse transform digital filter extending beyond the boundary, a two coefficient inverse transform digital filter is used:

$$D_0 = 4[(b-a)H_0 + (c-d)G_0] \quad (\text{equ. 29})$$

in the case where $K1 = aD_0$ and $K2 = dD_0$. D_0 is the first data value and H_0 is the data value to be inverse transformed adjacent the start boundary. This even start inverse transform digital filter has the form of the four coefficient even inverse transform digital filter except that the G_{-1} data value outside the boundary is chosen to

- 41 -

be equal to H_0 , and the H_{-1} data value outside the boundary is chosen to be equal to G_0 . The even start inverse transform digital filter therefore determines D_0 as a function of only H_0 and G_0 rather than as a function of H_{-1} , G_{-1} , H_0 and G_0 .

Similarly, a two coefficient odd end inverse transform digital filter is used to avoid the four coefficient odd inverse transform digital filter from extending beyond the end boundary at the other boundary of a sequence of data values to be inverse transformed. The two coefficient odd end inverse transform digital filter used is:

$$D_1 = 4[(c+d)H_1 - (a+b)G_1] \quad (\text{equ. 30})$$

in the case where $K4 = aD_1$ and $K3 = dD_1$. D_1 is the data value to be determined and G_1 is the data value to be inverse transformed adjacent the end boundary. This odd end inverse transform digital filter has the form of the four coefficient odd inverse transform digital filter except that the H_1 data value outside the boundary is chosen to be equal to G_1 and the G_1 data value outside the boundary is chosen to be equal to H_1 . The odd end inverse transform digital filter therefore determines D_1 as a function of only H_1 and G_1 rather than as a function of H_1 , G_1 , H_0 and G_0 .

It is to be understood that the particular even start and odd end inverse transform digital filters used in this embodiment are presented for illustrative purposes only. Where there is a different number of data values to be inverse transformed in a sequence of data values, an even end inverse transform digital filter may be used at the boundary rather than the odd end inverse transform digital filter. The even end inverse transform digital filter is an even inverse transform digital filter modified in accordance with the above process to have fewer coefficients than the even inverse transform digital

- 42 -

filter operating on the inner data values. Where filters other than quasi-Daubechies inverse transform digital filters are used, start and end inverse transform digital filters can be generated from the actual even and odd 5 inverse transform digital filters used to inverse transform data values which are not adjacent to a boundary. In the inverse transform, the start inverse transform digital filter processes the start of the transformed data values at the start boundary, then the 10 four coefficient inverse transform digital filters process the non-boundary transformed data values, and then the end inverse transform digital filter processes the end of the transformed data values.

The true Daubechies filter coefficients a , b , c and d 15 fulfil some simple relationships which show that the inverse transform digital filters correctly reconstruct non-boundary original image data values.

$$a+c = \frac{1}{2}, b-d = \frac{1}{2}, c+d = \frac{1}{4}, b-a = \frac{1}{4} \quad (\text{equ. 31})$$

and the second order equations:

$$20 \quad ac-bd = 0, a^2+b^2+c^2+d^2 = \frac{1}{2} \quad (\text{equ. 32})$$

Take two consecutive H,G pairs:

$$H\left(\frac{x}{2}\right) = aD(x-1)+bD(x)+cD(x+1)-dD(x+2) \quad (\text{equ. 33})$$

$$G\left(\frac{x}{2}\right) = dD(x-1)+cD(x)-bD(x+1)+aD(x+2) \quad (\text{equ. 34})$$

$$H\left(\frac{x}{2}+1\right) = aD(x+1)+bD(x+2)+cD(x+3)-dD(x+4) \quad (\text{equ. 35})$$

$$25 \quad G\left(\frac{x}{2}+1\right) = dD(x+1)+cD(x+2)-bD(x+3)+aD(x+4) \quad (\text{equ. 36})$$

Multiplying Equations 33 to 36 using the inverse transform digital filters gives:

- 43 -

$$cH\left(\frac{x}{2}\right) = acD(x-1) + bcD(x) + c^2D(x+1) - cdD(x+2) \quad (\text{equ. 37})$$

$$-bG\left(\frac{x}{2}\right) = -bdD(x-1) - bcD(x) + b^2D(x+1) - abD(x+2) \quad (\text{equ. 38})$$

$$aH\left(\frac{x}{2}-1\right) = a^2D(x+1) + abD(x+2) + acD(x+3) - adD(x+4) \quad (\text{equ. 39})$$

$$dG\left(\frac{x}{2}+1\right) = d^2D(x+1) + cdD(x+2) - bdD(x+3) + adD(x+4) \quad (\text{equ. 40})$$

$$5 \quad -dH\left(\frac{x}{2}\right) = -adD(x-1) - bdD(x) - cdD(x+1) + d^2D(x+2) \quad (\text{equ. 41})$$

$$aG\left(\frac{x}{2}\right) = adD(x-1) + acD(x) - abD(x+1) + a^2D(x+2) \quad (\text{equ. 42})$$

$$bH\left(\frac{x}{2}+1\right) = abD(x+1) + b^2D(x+2) + bcD(x+3) - bdD(x+4) \quad (\text{equ. 43})$$

$$cG\left(\frac{x}{2}+1\right) = cdD(x+1) + c^2D(x+2) - bcD(x+3) + acD(x+4) \quad (\text{equ. 44})$$

Summing equations 37-40 and 41-44 yields:

$$10 \quad cH\left(\frac{x}{2}\right) - bG\left(\frac{x}{2}\right) + aH\left(\frac{x}{2}+1\right) + dG\left(\frac{x}{2}+1\right) = \\ (ac-bd)D(x-1) + (a^2+b^2+c^2+d^2)D(x+1) + (ac-bd)D(x+3) = D(x+1)/2 \quad (\text{equ. 45})$$

$$-dH\left(\frac{x}{2}\right) + aG\left(\frac{x}{2}\right) + bH\left(\frac{x}{2}+1\right) + cG\left(\frac{x}{2}+1\right) = \\ (ac-bd)D(x) + (a^2+b^2+c^2+d^2)D(x+2) + (ac-bd)D(x+4) = D(x+2)/2 \quad (\text{equ. 46})$$

Using the coefficients of the four coefficient true Daubechies filter, the relationships of equations 31 and 32 hold. Equations 45 and 46 therefore show that with a one bit shift at the output, the original sequence of data 20 values is reconstructed.

Similarly, that the even start reconstruction filter of equation 29 and the odd end reconstruction filter of equation 30 correctly reconstruct the original image data adjacent the boundaries is shown as follows.

25 For the even start filter, with the choice of $K1 = aD_0$ and $K2 = dD_0$ in equations 29 and 30, we have:

- 44 -

$$H_0 = (a+b)D_0 + cD_1 - dD_2 \quad (\text{equ. 47})$$

$$G_0 = (c+d)D_0 - bD_1 + aD_2 \quad (\text{equ. 48})$$

so

$$bH_0 = b(a+b)D_0 + cbD_1 - dbD_2 \quad (\text{equ. 49})$$

$$5 \quad cG_0 = c(c+d)D_0 - cbD_1 + acD_2 \quad (\text{equ. 50})$$

$$aH_0 = a(a+b)D_0 + acD_1 - adD_2 \quad (\text{equ. 51})$$

$$dG_0 = d(c+d)D_0 - dbD_1 + adD_2 \quad (\text{equ. 51'})$$

and hence: from equation 29:

$$bH_0 + cG_0 - aH_0 - dG_0 = (b^2 - a^2 + c^2 - d^2)D_0 = \frac{D_0}{4} \quad (\text{equ. 52})$$

10 For the odd end filter, with the choice of $K_1 = dD_B$
and $K_4 = aD_B$, we have:

$$H_1 = aD_1 + bD_A + (c-d)D_B \quad (\text{equ. 53})$$

$$G_1 = dD_1 + cD_A + (a-b)D_B \quad (\text{equ. 54})$$

$$cH_1 = acD_1 + bcD_A + c(c-d)D_B \quad (\text{equ. 55})$$

$$15 \quad -bG_1 = -bdD_1 - bcD_A - b(a-b)D_B \quad (\text{equ. 56})$$

$$dH_1 = daD_1 + bdD_A + d(c-d)D_B \quad (\text{equ. 57})$$

$$-aG_1 = -adD_1 - caD_A - a(a-b)D_B \quad (\text{equ. 58})$$

and hence from equation 30:

$$(c+d)H_1 - (a+b)G_1 = (c^2 - d^2 + b^2 - a^2)D_B = \frac{D_B}{4} \quad (\text{equ. 59})$$

- 45 -

This reveals that the start and end boundary inverse transform digital filters can reconstruct the boundary data values of the original image when low pass and high pass start and end digital filters are used in the forward 5 transform.

TREE ENCODING AND DECODING

As described above, performing the forward quasi-perfect inverse transform does not reduce the number of data values carrying the image information. Accordingly, 10 the decomposed data values are encoded such that not all of the data values need be stored or transmitted. The present invention takes advantage of characteristics of the Human Visual System to encode more visually important information with a relatively larger number of bits while 15 encoding less visually important information with a relatively smaller number of bits.

By applying the forward quasi-perfect inverse transform to a two-dimensional array of image data values, a number of sub-band images of varying dimensions and 20 spectral contents is obtained. If traditional sub-band coding were used, then the sub-band images would be encoded separately without reference to each other except perhaps for a weighting factor for each band. This traditional sub-band encoding method is the most readily- 25 recognized encoding method because only the spectral response is accurately localized in each band.

In accordance with the present invention, however, a finite support wavelet is used in the analysis of an image, so that the sub-bands of the decomposition include 30 spatially local information which indicate the spatial locations in which the frequency band occurs. Whereas most sub-band encoding methods use long filters in order to achieve superior frequency separation and maximal stop band rejection, the filter used in the present invention 35 has compromised frequency characteristics in order to maintain good spatial locality.

- 46 -

Images can be thought of as comprising three components: background intensities, edges and textures. The forward quasi-perfect inverse transform separates the background intensities (the low pass luminance and chrominance bands) from the edge and texture information contained in the high frequency bands. Ideally, enough bandwidth would be available to encode both the edges and the textures so that the image would reconstruct perfectly. The compression due to the encoding would then be entirely due to removal of redundancy within the picture. If, however, the compressed data is to be transmitted and/or stored at low data transmission rates, some visual information of complex images must be lost. Because edges are a visually important image feature, the encoding method of the present invention locates and encodes information about edges or edge-like features for transmission or storage and places less importance on encoding textural information.

There are no exact definitions of what constitutes an edge and what constitutes texture. The present invention uses a definition of an edge that includes many types of textures. An edge or an edge-like feature is defined as a spatially local phenomenon giving rise to a sharp discontinuity in intensity, the edge or edge-like feature having non-zero spectral components over a range of frequencies. Accordingly, the present invention uses a frequency decomposition which incorporates spatial locality and which is invertible. The wavelet transform realized with quasi-perfect inverse transform digital filters meets these requirements.

Because an edge has non-zero components over a range of frequencies of the decomposition in the same locality, an edge can be located by searching through the wavelet decomposition for non-zero data values that represent edges. The method begins searching for edges by examining the low frequency sub-bands of the decomposition. These bands have only a small number of data values because of

- 47 -

the subsampling used in the wavelet transform and because the spatial support of each low frequency data value is large. After a quick search of the lowest frequency sub-bands, the positions of potential edges are determined.

5 Once the locations of the edges are determined in the lowest frequency sub-bands, these locations can be examined at a higher frequency resolutions to confirm that the edges exist and to more accurately determine their spatial locations.

10 Figure 23 illustrates an example of a one-dimensional binary search. There are three binary trees arranged from left to right in the decomposition of Figure 23. There are three octaves, octaves 0, 1 and 2, of decomposed data values in Figure 23. The low pass component is not
15 considered to be an octave of the decomposition because most of the edge information has been filtered out. Figures 24A-24D illustrate the forward transformation of a one-dimensional sequence of data values D into a sequence of transformed data values such as the tree structure of
20 Figure 23. The data values of the sequence of Figure 24A are filtered into low and high frequency components H and G of Figure 24B. The low frequency component of Figure 24B is then filtered into low and high frequency components HH and HG of Figure 24C. The low frequency
25 component HH of Figure 24C is then filtered into low and high frequency components HHH and HHG. The transformed data values of HHH block 240 of Figure 24D correspond with the low frequency component data values A, G and M of Figure 23. The transformed data values of HHG block 241
30 of Figure 24D correspond with the octave 2 data values B, H and N of Figure 23. The transformed data values of HG block 242 of Figure 24D correspond with the octave 1 data values of Figure 23. Similarly, the transformed data values of G block 243 correspond with the octave 0 data
35 values of Figure 23. Although only three trees are shown in Figure 23, the number of HHH data values in block 240 can be large and the size of the tree structure of Figure

- 48 -

23 can extend in the horizontal dimension in a corresponding manner.

The encoding of a one dimensional wavelet decomposition such as the decomposition of Figure 23 is performed in similar fashion to a binary tree search. The spatial support of a given data value in a given frequency band is the same as two data values in the octave above it in frequency. Thus the wavelet decomposition is visualized as an array of binary trees such as is illustrated in Figure 23, each tree representing a spatial locality. The greater the number of transform octaves, the higher the trees extend upward and the fewer their number.

As illustrated in Figure 23, each of the data values of the decomposition represents a feature which is either "interesting" to the human visual system, or it represents a feature that is "non-interesting" to the human visual system. A data value representing an edge of an object in an image or an edge-like feature is an example of an "interesting" data value. The encoding method is a depth first search, which starts at the trunk of a tree, ascends up the branches of the tree that are interesting, and terminates at the non-interesting branches. After all the branches of a tree have been ascended until a non-interesting data value is encountered or until the top of the branch is reached, the encoding of another tree is begun. Accordingly, as the encoding method follows the interesting data values of Figure 23 from octave 2 to octave 1 to octave 0, the edge is followed from low to high frequency resolution and an increasingly better approximation to the spatial position and shape of the edge is made. Conversely, if at any stage, a non-interesting data value is found, the search is terminated for data values above that non-interesting data value. The higher frequency data values of the tree above a non-interesting data value are assumed to be non-interesting because the corresponding low frequency data

- 49 -

values did not indicate the presence of an edge at this location. Any interesting data values that do exist in the higher frequency bands above a non-interesting data value in a low frequency band are rejected as noise.

5 The one-dimensional tree structure of Figure 23 is encoded as follows. The low frequency components carry visually important information and are therefore always considered to be "interesting". The method of encoding therefore starts with low frequency component A. This
10 data value is encoded. Next, the octave 2 data value B is tested to determine if it represents an edge or an edge-like feature which is "interesting" to the human visual system. Because data value B is interesting, a token is generated representing that the bits to follow will
15 represent an encoded data value. Interesting data value B is then encoded. Because this tree has not yet terminated, the method continues upward in frequency. Data value C of octave 1 is then tested. For purpose of this example, data value C is considered to be interesting
20 as are data values A, B, C, D, G, H, J, L and M as illustrated in Figure 23. A token is therefore generated indicating an encoded data value will follow. After the token is sent, data value C is encoded. Because this branch has still not terminated in a non-interesting data
25 value, the method continues upward in frequency. Data value D is tested to determine whether or not it is interesting. Because data value D is interesting, a token is generated and data value D is encoded. Because octave 0 is the highest octave in the decomposition, the encoding
30 method tests the other branch originating from previous interesting data value C. Data value E however tests to be non-interesting. A non-interesting token is therefore generated. Data value E is not encoded and does not appear in the compressed data. With both branches
35 originating at data value C terminated, the method proceeds down in frequency to test the remaining branches originating from the previous interesting data value B.

- 50 -

Data value F is, however, determined to be non-interesting. A non-interesting token is therefore generated and data value F is not encoded and does not appear in the encoded data. Because this branch has
5 terminated, all data values higher in frequency above data value F are considered to be non-interesting. A decoding device receiving the sequence of encoded data values and tokens can determine from the non-interesting token that all corresponding higher frequency data values were
10 considered to be non-interesting by the encoding device. The decoding device can therefore write the appropriate data values as non-interesting and write zeroes to these locations obviating the need for the encoding device to transmit each non-interesting data value above F. With
15 the first tree encoded, the method proceeds to the next low frequency component, data value G. This is a low frequency component and therefore is always considered to be interesting. Data value G is therefore encoded. The method then proceeds to the next tree through blocks H, I,
20 J, K and L in that order generating interesting and non-interesting tokens and encoding interesting data values. Similarly, after the second tree is terminated, low frequency component data value M is encoded. Data value N is determined to be non-interesting so a non-interesting
25 token is sent and the encoding of the third tree is terminated.

In accordance with another embodiment of the present invention, a two-dimensional extension of the one-dimensional case is used. Rather than using binary trees,
30 four branch trees are used. However, to create a practical image encoding method there are also real world factors to take into account. Using a single data value to predict whether the remainder of the tree is zero, is unreliable when dealing with noisy image data. A small
35 two-by-two block of data values is therefore used as the node element in the tree structure of the two-dimensional embodiment. A decision as to whether or not an edge is

- 51 -

present is based on four data values which is more reliable than a decision based on single data value.

Figure 25 illustrates a tree structure representing a portion of the decomposition of Figure 18. The decomposition of Figure 18 may extend farther to the right and farther in a downward direction for larger two-dimensional arrays of image data values. Similarly, the tree structure of Figure 25 may extend farther to the right for larger arrays of data values. Figure 25 represents a decomposition only having octave 0 and 1 high frequency components. In the event that the decomposition had additional octaves of high frequency components, the tree structure would extend further upward. In contrast to the binary tree structure of Figure 23, the tree structure of Figure 25 is a four branch tree. The two-by-two block of four octave 1 data values HHHG is the root of a tree which extends upward in frequency to four HG two-by-two blocks. If another octave of decomposition were performed, another level of octave 2 high frequency two-by-two blocks would be inserted into the tree structure. Four HHHG octave 1 two-by-two blocks would, for example, have a single octave 2 HHHHHG block beneath them. The low frequency component would be denoted HHHHHH.

Figure 26 is a pictorial representation of the decomposition of the tree structure of Figure 25. As explained above with respect to Figure 15, the actual data values of the various denoted blocks are distributed throughout the two-dimensional array of data values. The two numbers separated by a comma in each of the boxes of Figure 25 denote the row and column of a data value of the two-dimensional array of Figure 18, respectively. Using this tree structure, it is possible to search through the transformed data values of Figure 18 encoding interesting two-by-two blocks of data values and ignoring non-interesting two-by-two blocks.

To describe how the two dimensional encoding method uses the tree structure to search through a decomposition,

- 52 -

some useful definitions are introduced. First an image *decomp* is defined with dimensions *WIDTH* by *HEIGHT* decomposed to number *OCTS* of octaves. A function *Access* is defined such that given some arguments, the function
 5 *Access* outputs the memory address of the specified data value in the wavelet decomposition *decomp*:

```
address = Access (oct, sub, x, y);
```

oct is the octave of the data value sought and is an integer value between 0 (the highest octave) and *OCTS*-1
 10 (the number of octaves of transformation *OCTS* minus one). *sub* indicates which of the *HH*, *HG*, *GH* or *GG* bands of the decomposition it is that the data value sought is found. The use of *sub* = *HH* to access the low pass data values is only valid when the value of *oct* is set to that of the
 15 lowest octave. The co-ordinates *x* and *y* indicate the spatial location from the top left hand corner of the sub-band specified by *oct* and *sub*. The range of valid values of *x* and *y* are dependent on the octave being accessed. *x* has a range of {0. . . $WIDTH/2^{oct+1}$ }. *y* has a range of {0 .
 20 . . $HEIGHT/2^{oct+1}$ }.

Given the function *Access* and a wavelet decomposition, a two-by-two block of data values can be read by the function *ReadBlock*.

```
block = ReadBlock (decomp, oct, sub, x, y) {  

  25   block[0][0] = decomp[Access(oct, sub, x, y)];  

      block[0][1] = decomp[Access(oct, sub, x+1, y)];  

      block[1][0] = decomp[Access(oct, sub, x, y+1)];  

      block[1][1] = decomp[Access(oct, sub, x+1, y+1)];  

  }
```

30 The wavelet decomposition is passed to the function *ReadBlock* via the variable *decomp*. The two-by-two block of data values is returned through the variable *block*.

Once a two-by-two block of data values is read, a

- 53 -

decision is made as to whether the two-by-two block is visually "interesting" and should therefore be encoded or whether it is not and hence should be discarded. The decision is made by a function called *Threshold*. The 5 arguments of the function *Threshold* are *block*, *oct* and *sub*. *Threshold* returns a boolean value *True* if the block is "interesting" and *False* if the block is "non-interesting".

If the block is determined to be interesting by the 10 function *threshold*, it is encoded using a function called *EncodeBlock*. A function *SendToken* inserts a token before the encoded block to inform a decoding device which will later decode the compressed data whether the block to follow the token has been encoded (i.e. *BlockNotEmpty*) or 15 has not been encoded (i.e. *BlockEmpty*). If a block is determined to be interesting, then a *BlockNotEmpty* token is sent, and the block is encoded; next the tree structure above the encoded block is ascended to better determine the location of the edge. The tree encoding procedure 20 *SendTree* is therefore defined recursively as follows:

```

SendTree (decomp, oct, sub, x, y, Q) {
    block = ReadBlock (decomp, oct, sub, x, y);
    If Threshold (block, oct, sub, Q) {
        SendToken (BlockNotEmpty);
    25    EncodeBlock (block, oct, sub, Q);
        If (oct > 0) {
            SendTree (decomp, oct-1, sub, 2*x, 2*y, Q);
            SendTree (decomp, oct-1, sub, 2*(x+1), 2*y, Q);
            SendTree (decomp, oct-1, sub, 2*x, 2*(y+1), Q);
    30    SendTree (decomp, oct-1, sub, 2*(x+1), 2*(y+1), Q);
        }
    } else SendToken (BlockEmpty);
}

```

The procedure *SendTree* is only used to encode high- 35 pass component data values. In procedure *SendTree*

- 54 -

(*decomp*, *oct*, *sub*, *x*, *y*, *Q*), if the two-by-two block accessed by *ReadBlock* is determined to pass the threshold test, then *SendTree* (*decomp*, *oct*-1, *sub* 2*X, 2*y, *Q*) is used to test one of the next higher two-by-two blocks in the decomposition tree.

The low-pass data values are not considered to form part of the tree structure. The low-pass data values are encoded using another procedure *SendLPF*. In addition, the low-pass values are encoded using a different technique than that used in *EncodeBlock*, so a new procedure *EncodeBlockLPF* is required.

```

SendLPF (decomp, x, y, Q) {
    block = Readblock (decomp, OCTS-1, HH, x, y);
    EncodeBlockLPF (block, OCTS-1, Q);
15 }
```

Accordingly, to encode the entire image, *SendLPF* is applied to all the block locations within the low pass band and *SendTree* is applied to the all the block locations in the HG, GH and GG bands, within the lowest 20 octave. A procedure *SendDecomp* is therefore defined that encodes the entire image decomposition:

```

SendDecomp (decomp, Q) {
    For (y=0; y<HEIGHT/2OCTS; y=y+2)
    For (x=0; x<WIDTH/2OCTS; x=x+2) {
25     SendLPF (decomp, x, y, Q);
        SendTree (decomp, OCTS-1, HG, x, y, Q);
        SendTree (decomp, OCTS-1, GH, x, y, Q);
        SendTree (decomp, OCTS-1, GG, x, y, Q);
    }
30 }
```

Accordingly, the above functions define a method for encoding wavelet decomposed images. In terms of speed of encoding for real-world images, many of the trees are

- 55 -

terminated within the initial octaves so much of the decomposition is not examined. Due to this termination of many trees in the initial octaves, many data values need not be encoded which results in reducing the memory

5 bandwidth and block processing required to implement the compression/decompression method. Provided the functions *Threshold*, *EncodeBlockLPP* and *Access* require only simple calculations, the decomposed data values are rapidly encoded.

10 To implement the function *Access*, a table containing all the addresses of the data values of the two-dimensional tree decomposition may be accessed using the variables *x*, *y*, *sub* and *oct*. For a small image having a small number of data values, this table lookup approach is
15 reasonable. For images having, for example, approximately 80 different values of *x*, 60 different values of *y*, four different values of *sub*, and 3 or 4 values for *oct*, this table would contain approximately 150,000 10-bit locations. A less memory intensive way of determining the
20 same *X* and *Y* addresses from the same variables is desirable.

In accordance with one embodiment of the present invention, a function is used to determine the *X* and *Y* addresses from the variables *x*, *y*, *sub* and *oct*. Address
25 *X*, for example, may be determined as follows:

$$X = ((x \ll 1) + (sub \gg 1)) \ll oct$$

where \ll denotes one shift to the right of value *x* and
where \gg denotes one shift to the left.

Address *Y*, for example, may be determined as follows:

30
$$Y = ((y \ll 1) + (1 \& sub)) \ll oct$$

where $\&$ denotes a bit-wise AND function.

In a high performance system, the function *Access* may be implemented according to the following method. The

- 56 -

recursive function call and the table lookup methods described above are often too slow to implement in real time software or in hardware. Figures 27 and 28 illustrate how the tree decomposition of Figure 25 is traversed in order to generate tokens and encode two-by-two blocks of data values. The X and the Y in Figures 27 and 28 denote coordinate addresses in the two-dimensional matrix of Figure 18. In order to traverse the tree of the decomposition of Figure 25, it is necessary to be able to determine the X and Y addresses of the data values represented in Figure 25. Figure 27 illustrates how the X and Y address of a two-by-two block of data values are determined for those two-by-two blocks of data values located in octave 0 of the decomposition of Figure 25. Similarly, Figure 28 illustrates how the X and Y addresses of the three two-by-two blocks of data values in octave 1 of the decomposition as well as the one two-by-two block of data values of the low pass component of the decomposition of Figure 25 are determined. X as well as Y are each functions of oct, TreeRoot, and sub. The values of sub_h and sub_v are determined by the sub-band of the two-by-two block of data values sought.

Figure 29 is a chart illustrating the values of sub_h and sub_v for each sub-band of the decomposition. If, for example, a two-by-two block of data values is sought in the HH band, then the values of sub_h and sub_v are 0 and 0, respectively. The values TreeRoot_h and TreeRoot_v together denote the particular tree of a decomposition containing the particular two-by-two block of the data values sought.

In Figures 27 and 28, the rectangles represent digital counters. The arrows interconnecting the rectangles indicate a sequence of incrementing the counters. For example, the right most rectangle in Figure 27, which is called counter C1, has a least significant bit represented in Figure 27 as bit C1_l, and a most significant bit represented as bit C1_h. Similarly, the next rectangle to the left in Figure 27 represents a

- 57 -

digital counter C2 having two bits, a least significant bit C2₁, and a most significant bit C2₂. The structure of the X, Y address depends on the octave in which the two-by-two block of data values being sought resides. To
5 generate the X, Y address in octave oct = 1, the counter C1 is not included, the sub₁ and sub₂ bits indicating the sub-band bits are shifted one place to the left, and the least significant bits are filled with zeros. The
incrementing of the counters in Figure 28 proceeds as
10 illustrated by the arrows.

To determine the X and Y addresses of the four data values of the low pass component HHHH of Figure 25, Figure 28 is used. Because the two-by-two block of data values being sought is a two-by-two block of the low pass
15 component, the values of sub₁ and sub₂ are 0, 0 as required by the table of Figure 29. The C2 counter of Figure 28 increments through the four possible values of C2₁ and C2₂ to generate the four addresses in the two-by-two block of data values of the HHHH in the low pass component of
20 Figure 25. The value of TreeRoot₁ and TreeRoot₂ are zeroes because this is the first tree of the decomposition. For subsequent trees of the decomposition, TreeRoot₁ and TreeRoot₂ are incremented as illustrated by the arrows in Figure 28 so that the X and Y addresses of the other two-
25 by-two blocks of data values in the low pass component of the tree decomposition can be determined. After this HHHH two-by-two block of data values is located, the four data values are encoded and the search through the tree structure proceeds to the two-by-two block of data values
30 in octave 1 denoted HHHG in Figure 25. To determine the X and Y addresses of the four data values of this two-by-two block, the value of bits sub₁ and sub₂ are changed in accordance with Figure 29. Because this two-by-two block is in the HG sub-band, the values of sub₁ and sub₂ are 0
35 and 1, respectively. The C2 counter is then incremented through its four values to generate the four addresses of the four data values in that block. Supposing, that this

- 58 -

two-by-two block is determined to be "interesting" then an interesting token is sent, each of the four data values of the block are encoded, and the tree is then ascended to the two-by-two block of data values in octave 0 denoted

5 HG#1. These four addresses are determined in accordance with Figure 27. Because the sub-band is sub-band HG, the values of the bits sub_x and sub_y are 0 and 1, respectively. Counter C1 is then incremented so that the four addresses illustrated in the two-by-two block octave 0 HG#1 of

10 Figure 25 are generated. If the two-by-two block is interesting, then the interesting token is sent and the four data values are encoded. If the two-by-two block is determined not to be interesting, then a non-interesting token is sent and the four data values are not encoded.

15 The search through the tree structure of the decomposition then proceeds to octave 0 block HG#2. After the four addresses of the octave 0 block HG#1 are generated, the C2, bit of the C2 counter is incremented in accordance with the arrows shown in Figure 27. Accordingly, the octave 0

20 block HG#2 is addressed when once again the C1 counter increments through its four states. If the data values of this two-by-two block are determined to be "interesting", an interesting token is sent followed by the encoded data values. If the data values of the two-by-two block are

25 determined to be non-interesting, then a non-interesting token is sent. After all the search of the four two-by-two blocks of the octave 0 HG sub-band are searched, then that HG tree is terminated and the search proceeds to determine the four addresses of the four data values of

30 the octave 1 HHGH two-by-two block. In accordance with this technique, it is possible to traverse the structure of the decomposition and determine the addresses of any two-by-two block in any octave or any sub-band with minimum overhead. Moving between consecutive addresses or

35 descending trees is a simple operation when compared to the snaking address path used by other compression methods such as JPEG.

- 59 -

When implemented in software, this technique enables real time compression and decompression whereas other techniques may be too slow. If implemented in hardware, this technique provides for a reduced gate count and an efficient implementation. Although this example shows one way of traversing the tree structure of wavelet transform decomposition, it is possible to traverse the tree structure in other ways simply by changing the control structure represented in Figures 27 and 28 to allow for a different traversal of the tree structure. For example, all of the low pass HHHH blocks can be located and encoded first followed by all of the HHHG tree of the decomposition, and then all of the HHGH trees, and then all of the HHGG trees.

15

QUANTIZATION

Each data value of each two-by-two block of the tree decomposition which is determined to be "interesting" is quantized and then Huffman encoded. A linear mid-step quantizer with double-width-0 step is used to quantize each of the data values. Figure 30 is an illustration of the quantization of a 10-bit twos complement data value. The range of the 10-bit data value to be quantized ranges from -512 to 511 as illustrated by the numbers above the horizontal line in Figure 30. This range is broken up into a plurality of steps. Figure 31 represents one such step of data values which extends from 128 to 256 in Figure 30. All incoming data values having values between 128 and 255 inclusive are quantized by dividing the data value by the value $qstep$. Accordingly, the data value A having a value of 150 as illustrated in Figure 31 is divided by the $qstep$ value 128 and results in a $qindex$ number of 1. Integer division is used to generate $qindex$ and the fractional part of the remainder is discarded. Once the $qindex$ number is determined, the $qindex$ number is Huffman encoded. An overall Q value is sent once per frame of compressed data values. The value $qstep$ is

- 60 -

determined from the overall Q value as described below.

To inverse quantize the $qindex$ number and the $qstep$ value to determine the value of the transformed data values before inverse transformation, the device decoding the incoming quantized values calculates the value of $qstep$ using the value of Q according to a method described below. Once the value of $qstep$ is determined, $qindex$ for a given data value is multiplied by $qstep$.

In the example of Figure 31, $qindex$ value 1 times $qstep$ 128 results in an inverse quantized value of 128.

If this inverse quantized value of 128 were used, however, all the data values in the step 128 through 255 would be inverse quantized to the value of 128 at the left end of the step. This would result in unacceptably large errors. On the other hand, if all the data values in the range of Figure 31 were inverse quantized to the mid-step value 191, then less error would result. Accordingly, an inverse quantized value $qvalue$ can be calculated from $qindex$ and $qstep$ as follows:

$$qvalue(qindex, qstep) = \begin{cases} qindex \cdot qstep - \left(\frac{qstep}{2} - 1 \right) & \text{if } qindex < 0 \\ 0 & \text{if } qindex = 0 \\ qindex \cdot qstep + \left(\frac{qstep}{2} - 1 \right) & \text{if } qindex > 0 \end{cases}$$

The human visual system, however, has different sensitivities to quantization errors depending upon the particular sub-band containing the quantized data values. The human visual system performs complex non-linear processing. Although the way the human visual system relates image intensities to recognizable structures is not well understood, it is nevertheless important to take advantage of as much information about the human visual system as possible in order to maximize compression ratio versus picture quality. The wavelet transform approximates the initial image processing performed by the human brain. Factors such as spatial frequency response and Weber's Law can therefore be applied directly to the

- 61 -

wavelet transformed data values because the transformed data values are in a convenient representation.

Figure 32 shows the sensitivity of the human eye to spatial frequency. Spatial frequency is measured in 5 cycles c per visual angle θ . A screen is positioned at a distance d from an observer as illustrated in Figure 33. A light of sinusoidally varying luminance is projected onto the screen. The spatial frequency is the number of luminance cycles c per visual degree θ at distance d .

10 Note from Figure 32 that the sensitivity of the human eye varies with spatial frequency. Accordingly, the value of $qstep$ is varied depending on the octave and sub-band of the data value being quantized. The $qstep$ at which a data value is quantized is determined from the variables

15 oct , sub and Q for that data value as follows:

$$qstep(oct, sub, Q) = Q * hvs_factor(oct, sub)$$

$$hvs_factor(oct, sub) = \begin{cases} \sqrt{2} & \text{if } sub=GG \\ 1 & \text{otherwise} \end{cases} * \begin{cases} 1.00 & \text{if } oct=0 \\ 0.32 & \text{if } oct=1 \\ 0.16 & \text{if } oct=2 \\ 0.10 & \text{if } oct=3 \end{cases}$$

The scaling factors 1.00, 0.32, 0.16 and 0.10 relate to the spatial frequency scale of Figure 32 to take into

20 account the frequency dependent sensitivity of the human eye.

It is to be understood that scaling factors other than 1.00, 0.32, 0.16 and 0.10 could be used. For example, other scaling factors can be used where the

25 quantizer is used to compress audio data which is received by the human ear rather than by the human eye. Moreover, note that the sub-band GG is quantized more heavily than the other sub-bands because the sub-band GG contains diagonal information which is less important to the human

30 eye than horizontal and vertical information. This method can also be extended down to the level of two-by-two blocks of data values to further tailor the degree of quantization to the human visual system. The function

- 62 -

hvs_factor which has only two parameters in the presently described embodiment is only one embodiment of the present invention. The function *hvs_factor*, for example, can take into account other characteristics of the human visual system other than *oct* and *sub*, such as the luminance of the background and texture masking.

THRESHOLDING

For each new two-by-two block of data values in the tree decomposition, a decision must be made as to whether the block is "interesting" or "non-interesting". This can be done by the function *threshold*:

$$\text{threshold}(\text{block}, \text{limit}) = \text{limit} > \sum_{y=0}^1 \sum_{x=0}^1 |\text{block}[y][x]| \quad (\text{equ. 60})$$

The sum of the absolute values of the data values of the block *block* is determined as is represented by the double summation to the right of the less than sign and this value is compared to a threshold value *limit*.

"Interesting" blocks are those blocks, for which the sum of the absolute values of the four data values exceeds the value *limit*, whereas "non-interesting" blocks are those blocks for which the sum is less than or equal to the value *limit*.

The value *limit* takes into account the variable quantizer step size *qstep* which varies with octave. For example, a two-by-two block of data values could be determined to pass the test *threshold*, but after quantizing by *qstep* could result in four zero quantized values. For example, all data values between -128 and 127 are quantized to have a quantized *qindex* of zero as is shown in Figure 30 even if some of those data values are determined to correspond with an "interesting" two-by-two block. For this reason, the value *limit* is calculated according to the equation:

- 63 -

$$\text{limit} = 4 * B_{\text{threshold}} * q_{\text{step}} \quad (\text{equ. 61})$$

In this equation "*Bthreshold*" is base threshold image factor. In the presently described example, this base threshold is equal to 1.0. The value of 1.0 for the base threshold *Bthreshold* was determined through extensive experimentation on test images. The factor 4 in equation 61 is included to account for the fact that there are four data values in the block under consideration. In this way blocks are not determined to be interesting, the data values for which the quantizer will later reduce to zeros. This weighted threshold factor *limit* also reduces the number of operations performed in the quantizer because a fewer number of data values are quantized.

HUFFMAN CODING

The wavelet transform produces transformed data values whose statistics are vastly different from the data values of the original image. The transformed data values of the high-pass sub-bands have a probability distribution that is similar to an exponential or Laplacian characteristic with mean zero.

Figure 34 shows the distribution of high pass data values in a four octave wavelet decomposition of the test image Lenna. Figure 35 shows the distribution of the data values of the test image Lenna before wavelet transformation. The low-pass component data values have a flat distribution that approximates the distribution of luminance and chrominance values in the original image. The high and low pass data values are encoded differently for this reason.

The low pass component data values are encoded by the function *EncodeBlockLPF* as follows:

```

EncodeBlockLPF ( block, OCT-1, Q) {
    Output ( block[0][0]/qstep( OCT-1, HH, Q));
    Output ( block[0][1]/qstep( OCT-1, HH, Q));
    Output ( block[1][0]/qstep( OCT-1, HH, Q));

```

- 64 -

```
Output ( block[1][1]/qstep( OCT-1, HH, Q));)
```

After encoding, the low-pass data values are quantized and output into the compressed data stream. The low pass data values are not Huffman encoded.

- 5 The high frequency component data values which pass the threshold test are quantized and Huffman encoded to take advantage of their Laplacian distribution. Function *EncodeBlock* performs the quantization and the Huffman encoding for each of the four data values of an
- 10 interesting high frequency component block *block*. In the function *EncodeBlock*, the variable *sub* is provided so that when function *qstep* is called, different quantization *qstep* values can be used for different high frequency component sub-bands. The function *huffman* performs a
- 15 table lookup to a fixed Huffman code table such as the table of Table 3. The function *EncodeBlock* is defined as follows:

```
EncodeBlock (block, oct, sub, Q) {  
    output(huffman(block[0][0]/qstep(oct, sub, Q)));  
20    output(huffman(block[0][1]/qstep(oct, sub, Q)));  
    output(huffman(block[1][0]/qstep(oct, sub, Q)));  
    output(huffman(block[1][1]/qstep(oct, sub, Q)));  
}
```

- 65 -

	<i>qindex</i>	Huffman code
	-38 . . . -512	1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1
	-22 . . -37	1 1 0 0 0 0 0 0 1 1 1 1 (<i> qindex </i> -22)
	-7 . . -21	1 1 0 0 0 0 0 0 (<i> qindex </i> -7)
5	-6	1 1 0 0 0 0 0 1
	.	.
	.	.
	.	.
	-2	1 1 0 1
10	-1	1 1 1
	0	0
	1	1 0 1
	2	1 0 0 1
	.	.
15	.	.
	.	.
	6	1 0 0 0 0 0 0 1
	7 . . 21	1 0 0 0 0 0 0 0 (<i> qindex </i> -7)
	22 . . 37	1 0 0 0 0 0 0 0 1 1 1 1 (<i> qindex </i> -22)
20	38 . . 511	1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

Table 3

The second bit from the left in the Huffman code of Table 3 is a sign bit. The value $|qindex|-7$ is represented with 4 bits in the case $7 \leq |qindex| \leq 21$. The value $|qindex|-22$ is represented with 4 bits in the case $22 \leq |qindex| \leq 37$.

ENCODING OF TOKENS

At high compression ratios the number of bits in the compressed data stream used by tokens may be reduced by amalgamating groups of "non-interesting" tokens. This can be achieved by introducing new tokens. In accordance with one embodiment of the present invention, two new tokens, *OctEmpty* and *OctNotEmpty* are used. For a high pass component block in a tree above octave zero, there are four branches. The additional pair of tokens indicate

- 66 -

whether all four are non-interesting. If all four are non-interesting, only a single *OctEmpty* token need be sent. Otherwise, an *OctNotEmpty* token is generated before the four branches are encoded. The particular token
5 scheme described above was selected more to simplify the hardware and software implementations than it was to achieve in the best compression ratio possible. Other methods of representing relatively long sequences of token
10 bits in the compressed data stream using other tokens having a relatively fewer number of bits may be used in place of the tokens *OctEmpty* and *OctNotEmpty* to achieve higher compression ratios.

VIDEO ENCODING AND DECODING

In comparison with the coding of a still image, the
15 successive images of a video sequence typically contain much redundant information. The redundancy of this information is used to reduce the bit rate. If a location in a new frame of the video contains the same or
substantially the same information as a corresponding
20 location in the previous old frame of video, that portion of the new frame need not be encoded and introduced into the compressed data. This results in a reduction in the total number of bits in the encoded bit stream.

Figure 36 illustrates a video encoder 31 and a video
25 decoder 32. A video input signal is transformed by a forward wavelet transform block 33, the output of which is written to a new frame store 34. The first frame of video information in the new frame store 34 is referred to as the new frame because no previous frame exists in the old
30 frame store 35 for containing an old frame. A comparison tree encoder 36 therefore generates tokens and transformed data values as described above from the data values output from new frame store 34. The transformed data values are quantized by quantizer 37 into *gindex* levels. These
35 *gindex* levels are then Huffman coded by the Huffman encoder 38. The resulting encoded data values are then

- 67 -

combined with the tokens in buffer 38A to form a decompressed data bit stream 39.

An essential part of this method is that the old frame present in the video encoder 31 is exactly the same as the old frame 40 present in the video decoder 32. This allows the decoder 32 to be able to correctly decode the encoded bit stream 39 due to the fact that the encoded bit stream contains differences between new and old images and due to the fact that parts of the new frame are not sent due to compression. An inverse quantizer 41 is therefore provided in the video encoder 31 to inverse quantize the *qindex* levels and to store the old frame as sent into old frame store 35 for future comparison with the next frame of the video input signal.

In the video decoder 32, the compressed data stream 39 is received by a buffer 42. The tokens are separated from the Huffman encoded *qindex* levels. The Huffman encoded *qindex* levels are supplied to a Huffman decoder 43, the output of which is supplied to an inverse quantizer 44. The output of the inverse quantizer 44 is written into old frame store 40 under the control of the comparison tree decoder 45. Comparison tree decoder 45 determines what is written into the old frame store 40, depending in part on the tokens received from buffer 42. Once a new frame of transformed data values is present in old frame store 40, an inverse wavelet transform 46 inverse transforms that frame of transformed data values into a corresponding video output signal. To prevent the inverse wavelet transform 46 from overwriting and therefore corrupting the contents of old frame store 40 when it reconstructs data values corresponding to the original new frame data values, an intermediate frame store 47 is maintained.

The octave one HHNG, HHGH, HHGG, and HHHH from Figure 25 are read from the old frame store 40 by the inverse wavelet transform 46 to perform the octave 1 inverse transform as described above. However, the resulting

- 68 -

octave 0 HH sub-band, output from the inverse wavelet transform 46 is now written to the intermediate frame store 47, so as not to corrupt the old frame store 40. For the octave 0 inverse wavelet transform, the HG, GH, and GG 5 sub-bands are read from the old frame store 40, and the HH sub-band is read from the intermediate frame store 47, to complete the inverse wavelet transform.

When the second frame of compressed video data 39 is received by the video decoder 32, the tokens received by 10 the comparison tree decoder 45 are related to the contents of the previous frame of video information contained in old frame store 40. Accordingly, the video decoder 32 can reconstruct the latest frame of video data using the contents of the frame store 40 and the data values encoded 15 in the compressed data stream 39. This is possible because the compressed data stream contains all the information necessary for the video decoder 32 to follow the same traversal of the tree of the decomposition that the encoder used to traverse the tree in the generation of 20 the compressed data stream. The video decoder 32 therefore works in lock step with the video encoder 31. Both the encoder 31 and the decoder 32 maintain the same mode at a corresponding location in the tree. When the encoder 31 determines a new mode, it incorporates into the 25 compressed data stream 39 a corresponding token, which the video decoder 32 uses to assume that new mode.

Figure 37 illustrates the modes of operation of one possible embodiment of the present invention. To explain the operation of the video encoder 31 and the video 30 decoder 32, an example is provided. The initial frame of the video sequence is processed by the video encoder 31 in still mode. Still mode has three sub-modes: STILL, VOID_STILL, and LPF_STILL. The low pass two-by-two blocks of data values of the decomposition cause the comparison 35 tree encoder 36 of video encoder 31 to enter the LPF_STILL sub-mode. In this sub-mode, the four data values of the two-by-two block are quantized but are not Huffman

- 69 -

encoded. Similarly, no token is generated. The successive low pass component two-by-two blocks of data values are successively quantized and output into the compressed data stream 39.

5 Next, the lowest frequency octave of one of the sub-bands is processed by the comparison tree encoder 36. This two-by-two block of data values corresponds with block HHHG illustrated in Figure 25. The four data values of this two-by-two block are tested against the threshold
10 limit to determine if it is "interesting". If the two-by-two block HHHG is interesting, then a single bit token 1 is generated, as illustrated in Figure 37, the mode of the comparison tree encoder remains in STILL mode, and the four data values of the two-by-two block HHHG are
15 successively quantized and encoded and output into the compressed data stream 39.

For the purposes of this example, block HHHG is assumed to be interesting. The tree structure of Figure 25 is therefore ascended to octave 0 two-by-two block
20 HG#1. Because the comparison tree encoder 31 remains in the STILL mode, this block is encoded in the STILL mode. The four data values of block HG#1 are tested to determine whether or not they are interesting. This sequence of testing the successive blocks of the tree structure is
25 repeated as described above.

After the traversal of the four octave 0 sub-blocks HG#1, HG#2, HG#3 and HG#4, the comparison tree encoder 36 proceeds in the tree structure to the two-by-two block of data values in octave 1, block HHGH. For purposes of this
30 example, this two-by-two is non-interesting. After the comparison tree encoder 36 reads the four data values, the result of the threshold test indicates a non-interesting two-by-two block. As illustrated in Figure 37, the encoder 31 which is in the still mode now generates a
35 single bit token 0 and the comparison tree encoder 36 enters the VOID_STILL sub-mode. Although no additional information is output into the compressed data stream 39,

- 70 -

the comparison tree encoder 36 proceeds to write 0's into the four locations of the two-by-two block HHGH, as well as all the locations of the two-by-two blocks in the tree above the non-interesting two-by-two block HHGH. In the 5 example of Figure 25, the comparison tree encoder 36 writes 0's into all the addresses of blocks HHGH, GH#1, GH#2, GH#3 and GH#4. This zeroing is performed because the video decoder 32 will not be receiving the data values corresponding to that tree. Rather, the video decoder 32 10 will be receiving only a non-interesting token, a single bit 0. The video decoder 32 will therefore write zeros into frame store 40 in the remainder of the corresponding tree. In order to make sure that both the video encoder 31 and the video decoder 32 have exactly the same old 15 frame 35 and 40, the video encoder too must zero out those non-interesting blocks.

After the first frame of video data has been encoded and sent in STILL mode, the next frame of video data is processed by the video encoder 31. By default, the 20 encoder now enters SEND mode. For lowpass frequency component two-by-two blocks, the video encoder 31 enters the LPF_SEND mode as illustrated in Figure 37. The encoding of such a lowpass component two-by-two block corresponds with the encoding of two-by-two block HHHH in 25 Figure 25. However, now the comparison tree encoder 36 has both a new frame in frame store 34 as well as an old frame in frame store 35. Accordingly, the comparison tree encoder 36 determines the arithmetic difference of the respective four data values in the new frame from the four 30 data values in the old frame at the corresponding position and compares the sum of those differences with a compare threshold. The compare threshold, compare, is calculated from a base compare threshold "Bcompare" as in the case of the previous threshold which determines which blocks are 35 interesting, similar to equations 60 and 61. If the sum of the differences is less than the compare threshold, then the video encoder 31 sends a single bit token 0 and

- 71 -

remains in the LPF_SEND mode, as illustrated in Figure 37. The video encoder 31 does not transmit any data values corresponding to the lowpass frequency component two-by-two block.

5 If, on the other hand, the sum of the arithmetic differences exceeds the compare threshold, then a single bit token 1 is generated, as illustrated in Figure 37. In this case, the video encoder 31 sends the arithmetic differences of each of the successive four data values of
10 the new frame versus the old frame to the quantizer 37 and then to the Huffman encoder 38. The arithmetic differences are encoded and sent rather than sending the actual data values because this results in fewer bits due to the fact that the two blocks in the new and old frames
15 are quite similar under normal circumstances.

When the video encoder 31 proceeds to encode the octave 1 sub-band HHHG, as illustrated in Figure 25, the video encoder 31 enters the SEND mode, as illustrated in Figure 37. In this mode, the comparison tree encoder 36
20 compares the data values of the new two-by-two block with the data values of the old two-by-two block and performs a series of arithmetic operations to generate a series of flags, as illustrated in Figure 38. Based on these flags, the video encoder 31 generates a 2-bit token and enters
25 one of four new modes for that two-by-two block. If, for example, the two-by-two block HHHG in Figure 25 is received by the video encoder 31, then flags ozflag, nzflag, new_z, noflag, motion, origin, and no_z are determined. The values of these flags are determined as:

$$30 \quad nz = \sum_{x=0}^1 \sum_{y=0}^1 |new[x][y]| \quad (\text{equ. 62})$$

$$no = \sum_{x=0}^1 \sum_{y=0}^1 |new[x][y] - old[x][y]| \quad (\text{equ. 63})$$

$$oz = \sum_{x=0}^1 \sum_{y=0}^1 |old[x][y]| \quad (\text{equ. 64})$$

- 72 -

nzflag = nz < limit (equ. 65)
 noflag = no < compare (equ. 66)
 origin = nz ≤ no (equ. 67)
 motion = ((nz + oz) << oct) ≤ no (equ. 68)
 5 new_z = |new[x][y]| < qstep, 0 ≤ x, y, ≤ 1 (equ. 69)
 no_z = |new[x][y] - old [x][y]| < qstep, 0 ≤ x, y ≤ 1 (equ. 70)
 ozflag = {old[x][y] = 0; for all 0 ≤ x, y ≤ 1} (equ. 71)

Based on the values of these flags, the new mode for
 10 the two-by-two block HHHG is determined, from Figure 38.

If the new mode is determined to be the SEND mode,
 the 2-bit token 11 is sent as indicated in Figure 37. The
 arithmetic differences of the corresponding four data
 values are determined, quantized, Huffman encoded, and
 15 sent into the compressed data stream 39.

In the case that the flags indicate the new mode is
 STILL_SEND, then the 2-bit token 01 is sent and the new
 four data values of the two-by-two block are quantized,
 Huffman encoded, and sent. Once having entered the
 20 STILL_SEND mode, the video encoder 31 remains in the
 STILL_SEND mode until the end of the tree has been
 reached. In this STILL_SEND mode, a single bit token of
 either 1 or 0 precedes the encoding of each block of data
 values. When the VOID mode is entered from STILL_SEND
 25 mode, the video encoder 31 generates a single bit 0 token,
 then places zeros in the corresponding addresses for that
 two-by-two block, and then proceeds to place zeros in the
 addresses of data values of the two-by-two blocks in the
 tree above.

30 In the event that the flags indicate that the video
 encoder 31 enters the VOID mode from SEND mode, a 2-bit
 token 10 is generated and the four data values of that
 two-by-two block are replaced with zeros. The VOID mode
 also results in the video encoder 31 placing zeros in all
 35 addresses of all data values of two-by-two blocks in the
 tree above.

In the case that the flags indicate that there is no

- 73 -

additional information in the tree being presently encoded, namely, the new and the old trees are substantially the same, then a 2-bit token of 00 is generated and the video encoder 31 proceeds to the next 5 tree in the decomposition.

In general, when the video encoder 31 enters VOID mode, the video encoder will remain in VOID mode until it determines that the old block already contains four zero data values. In this case, there is no reason to continue 10 in VOID mode writing zeros into that two-by-two block or the remainder of the blocks in the tree above because it is guaranteed that the old tree already contains zeros in these blocks. This is true because the old tree in frame store 35 has previously been encoded through the inverse 15 quantizer 41.

Because the video decoder 32 is aware of the tree structure of the decomposition, and because the video encoder 31 communicates with the video decoder 32 using tokens, the video decoder 32 is directed through the tree 20 structure in the same manner that the video encoder 31 traverses the tree structure in generating the compressed data stream 39. In this way the video decoder 32 writes the appropriate data values from the decompressed data stream 39 into the corresponding positions of the old data 25 frame 40. The only flag needed by the video decoder 32 is the ozflag, which the video decoder obtains by reading the contents of old frame store 40.

RATE CONTROL

All transmission media and storage media have a 30 maximum bandwidth at which they can accept data. This bandwidth can be denoted in terms of bits per second. A standard rate ISDN channel digital telephone line has, for example, a bandwidth of 64 kbits/sec. When compressing a sequence of images in a video sequence, depending upon the 35 amount of compression used to compress the images, there may be a relatively high number of bits per second

- 74 -

generated. This number of bits per second may in some instances exceed the maximum bandwidth of the transmission media or storage device. It is therefore necessary to reduce the bits per second generated to insure that the maximum bandwidth of the transmission media or storage device is not exceeded.

One way of regulating the number of bits per second introduced into the transmission media or storage device involves the use of a buffer. Frames having a high number of bits are stored in the frame buffer, along with frames having a low number of bits, whereas the number of bits per second passing out of the buffer and into the transmission media or storage device is maintained at a relatively constant number. If the buffer is sufficiently large, then it is possible to always achieve the desired bit rate as long as the overall average of bits per second being input into the buffer over time is the same or less than the maximum bit rate being output from the buffer to the transmission media or storage device.

There is, however, a problem associated with large buffers in video telephony. For a large buffer, there is a significant time delay between the time a frame of video data is input into the buffer and time when this frame is output from the video buffer and into the transmission media or storage device. In the case of video telephony, large buffers may result in large time delays between the time when one user begins to speak and the time when another user begins to hear that speech. This time delay, called latency, is undesirable. For this reason, buffer size is specified in the standard H.261 for video telephony.

In accordance with one embodiment of the present invention, a rate control mechanism is provided which varies the number of bits generated per frame, on a frame by frame basis. Due to the tree encoding structure described above, the number of bits output for a given frame is dependent upon the number of trees ascended in

- 75 -

the tree encoding process. The decisions of whether or not to ascend a tree are made in the lowest high frequency octaves of the tree structure. As can be seen from Figure 25, there are relatively few number of blocks in the 5 lowest frequency of the sub-bands, as compared to the number of blocks higher up in the sub-band trees. Given a particular two-by-two block in the tree structure, it is possible to decrease the value of Q in the equation for the threshold limit until that particular block is 10 determined to be "interesting". Accordingly, a particular Q is determined at which that particular block becomes interesting. This process can be done for each block in the lowest frequency HG, GH and GG sub-bands. In this way, a histogram is generated indicating a number of 15 two-by-two blocks in the lowest frequency of the three sub-bands which become interesting at each particular value of Q .

From this histogram, a relationship is developed of the total number of two-by-two blocks in the lowest 20 frequency of the three sub-bands which are interesting for a given value of Q . Assuming that the number of blocks in the lowest frequency octave of the three sub-bands which are interesting for a given value of Q is representative of the number of bits which will be generated when the 25 tree is ascended using that given value of Q , it is possible to determine the value of Q at which a desired number of bits will be generated when that frame is coded with that value of Q . Furthermore, the greater the threshold is exceeded, the more bits may be needed to 30 encode that tree. It is therefore possible to weight by Q the number of blocks which are interesting for a given value of Q . Finally, the Q values so derived should be averaged between frames to smooth out fluctuations.

The encoder model RM8 of the CCITT Recommendation 35 H.261 is based on the DCT and has the following disadvantages. The rate control method used by RM8 is a linear feedback technique. Buffer fullness is

- 76 -

proportional to Q . The value of Q must be adjusted after every group of blocks (GOB) to avoid overflow or underflow effects. This means that parts of the image are transmitted at a different level quality from other parts.

5 During parts of the image where little change occurs, Q drops which can result in uninteresting areas being coded very accurately. The objects of interest are, however, usually the moving ones. Conversely, during the coding of areas of high activity, Q rises creating large errors in

10 moving areas. When this is combined with a block based transform, the errors can become visually annoying.

The method of rate control described in connection with one embodiment of the present invention uses one value of Q for the whole frame. The value of Q is only

15 adjusted between frames. All parts of an image are therefore encoded with the same value of Q . Moreover, because the tree structure allows a relatively few number of blocks to be tested to determine an estimate of the number of bits generated for a given frame, more

20 intelligent methods of varying Q to achieve an overall desired bit rate are possible than are possible with conventional compression/decompression techniques.

TREE BASED MOTION ESTIMATION

Figure 39 represents a black box 1 on a white

25 background 2. Figure 40 represents the same black box 1 on the same white background 2 moved to the right so that it occupies a different location. If these two frames of Figures 39 and 40 are encoded according to the above described method, there will be a tree in the wavelet

30 decomposition which corresponds with the white-to-black edge denoted 3 in Figure 39. Similarly, there will be another tree in the wavelet decomposition of the image of Figure 40 which represents the white-to-black edge 3' the wavelet decomposition of the image of Figure 40. All of

35 the data values corresponding to these two trees will be determined to be "interesting" because edges result in

- 77 -

interesting data values in all octaves of the decomposition. Moreover, due to the movement of the corresponding edge of black box 1, all the data values of the edges of both of these two trees will be encoded as 5 interesting data values in the resulting compressed data stream. The method described above therefore does not take into account that it is the same data values representing the same white-to-black edge which is present in both images but which is just located at a different 10 location.

Figure 41 is a one dimensional representation of an edge. The corresponding low path component data values are not illustrated in Figure 41. Data values 4, 5, 6, 7, 8, and 9 represent the "interesting" data values of Figure 15 41 whereas the other data values have low data values which makes those blocks "non-interesting". In the representation of Figure 41, data values 4 and 5 are considered a single two data value block. Similarly, blocks 6 and 7 are considered a single block and blocks 8 20 and 9 are considered a single block. Figure 41, although it is a one dimensional representation for ease of illustration, represents the edge 3 of the frame of Figure 39.

Figure 42 represents the edge 3' shown in Figure 40. 25 Figure 42 indicates that the edge of black box 1 has moved in location due to the fact that the values 19 and 21 which in Figure 41 were in the two data value block 8 and 9 are located in Figure 42 in the two data value block 10 and 11. In the encoding of Figure 42, rather than 30 encoding and sending into the compressed data stream the values 19 and 21, a control code is generated which indicates the new locations of the two values. Although numerous control codes are possible, only one embodiment is described here.

35 When the two data value block 10 and 11 is tested to determine whether it is interesting or not, the block tests to be interesting. The neighboring blocks in the

- 78 -

old frame are, however, also tested to determine whether the same values are present. In this case, the values 19 and 21 are determined to have moved one two data value block to the right. An "interesting with motion" token is 5 therefore generated rather than a simple "interesting" token. A single bit 1 is then sent indicating that the edge represented by values 19 and 21 has moved to the right. Had the edge moved to the left, a control code of 0 would have been sent indicating that the edge 10 represented by values 19 and 21 moved one location to the left. Accordingly, in the encoding of Figure 42, an "interesting with motion" token is generated followed by a single control code 1. The interesting values 19 and 21 therefore need not be included in the compressed data 15 stream. The video decoder receiving this "interesting with motion" token and this control code 1 can simply copy the interesting values 19 and 21 from the old frame into the indicated new location for these values in the new frame obviating the need for the video encoder to encode 20 and transmit the actual interesting data values themselves. The same token and control codes can be sent for the two data values corresponding to a block in any one of the octaves 0, 1 or 2.

Figure 43 represents the motion of the edge 3 of 25 Figure 39 to a new location which is farther removed than is the new location of black box 1 shown in Figure 40. Accordingly, it is seen that the values 20 and 21 are located to the right at the two data value block 12 and 13. In the encoding of this two data value block 12 and 30 13 a token indicating "interesting with motion" is generated. Following that token, a control code 1 is generated indicating motion to the right. The video encoder therefore need not encode the data values 20 and 21 but merely needs to generate the interesting with 35 motion token followed by the motion to the right control code. When the video encoder proceeds to the two data values block 14 and 15, the video encoder need not send

- 79 -

the "interesting with motion" token but rather only sends the left control code 0. Similarly, when the video encoder proceeds to encode the two data value block 16 and 17, the video encoder only sends the left control code 0.

5 The control codes for octaves 0 and 1 do not denote motion per se but rather denote left or right location above a lower frequency interesting block of the moving edge. This results in the video encoder not having to encode any of the actual data values representing the moved edge in
10 the decomposition of Figure 43.

The one dimensional illustration of Figures 41, 42 and 43 is presented for ease of illustration and explanation. It is to be understood, however, that this method of indicating edge motion is used in conjunction
15 with the above described two dimensional wavelet decomposition such as the two dimensional wavelet decomposition illustrated in Figure 25. The video encoder searches for movement of the data values representing an edge only by searching the nearest neighboring blocks of
20 data values in the old frame. This method can be used to search many neighbors or a few neighbors depending on the application. The counter scheme described in connection with Figures 27 and 28 can be used to determine the locations of those neighboring blocks. Although the edge
25 motion illustrated in connection with Figures 41, 42, and 43 shows the very same data values being moved in the tree structure of the decomposition, it is to be understood that in practice the values of the data values representing the same edge may change slightly with the
30 movement of the edge. The video encoder takes this into account by judging corresponding data values using a motion data value threshold to determine if corresponding data values in fact do represent the same edge. By indicating edge motion and not sending the edge data
35 values themselves it is possible to both increase the compression and also improve the quality of the decompressed image.

- 80 -

SIX COEFFICIENT QUASI-DAUBECHIES FILTERS

The Daubechies six coefficient filters are defined by the six low pass filter coefficients, listed in the table below to 8 decimal places. The coefficients are also defined in terms of four constants, α , β , γ and ϵ , where $\alpha = 0.10588942$, $\beta = -0.54609641$, $\gamma = 2.4254972$ and $\epsilon = 3.0059769$.

	Daubechies coefficients	Alternative representation	Normalized coefficients	Converted Coefficients
10	a	$1/\epsilon$	0.2352336	$\frac{30}{128}$
	b	γ/ϵ	0.57055846	$\frac{73}{128}$
	c	$-\beta(\alpha+\gamma)/\epsilon$	0.3251825	$\frac{41}{128}$
	-d	$\beta(1 - \alpha\gamma)/\epsilon$	-0.095467208	$\frac{-12}{128}$
	-e	$-\alpha\gamma/\epsilon$	-0.060416101	$\frac{-7}{128}$
	f	α/ϵ	0.024908749	$\frac{3}{128}$

Table 4

15 The coefficients (a, b, c, -d, -e, f) sum to $\sqrt{2}$. The normalized coefficients sum to 1, which gives the filter the property of unity gain, which in terms of the alternative representation is equivalent to a change in the value of ϵ to 4.2510934. These values can be approximated to any given precision by a set of fractions. In the example shown above, each of the normalized values has been multiplied by 128 and rounded appropriately, thus the coefficient a has been converted to $\frac{30}{128}$. Filtering is therefore possible using integer multiplications rather than floating point arithmetic. This greatly reduces implementation cost in terms of digital hardware gate count and computer software speed. The following equations show a single step in the filtering process, the outputs H and G being the low and high pass outputs, respectively:

$$H_1 = aD_0 + bD_1 + cD_2 - dD_3 - eD_4 + fD_5 \quad (\text{equ. 72})$$

- 81 -

$$G_1 = -fD_0 - eD_1 + dD_2 + cD_3 - bD_4 + aD_5 \quad (\text{equ. 73})$$

H_1 and G_1 are calculated as follows. Each data value D is multiplied by the relevant integer numerator (30, 73, 41, 12, 7, 3) and summed as shown. The values of H and G are found by dividing the summations by the constant 128. Because 128 is an integer power of 2, the division operation requires little digital hardware to implement and only simple arithmetic shift operations to implement in software. The filters H and G are quasi-perfect reconstruction filters:

$$a+b+c-d-e+f=1 \quad (\text{equ. 74})$$

$$-f-e+d+c-b+a=0 \quad (\text{equ. 75})$$

$$a+c-e=\frac{1}{2} \quad (\text{equ. 76})$$

$$f-d+b=\frac{1}{2} \quad (\text{equ. 77})$$

Equation 74 guarantees unity gain. Equation 75 guarantees that the high pass filter will generate zero for a constant input signal. Equations 76 and 77 guarantee that an original signal once transferred can be reconstructed exactly.

The following equations show a single step in the inverse transformation:

$$D_2 = 2(-eH_0 - bG_0 + cH_1 + dG_1 + aH_2 - fG_2) \quad (\text{equ. 78})$$

$$D_3 = 2(fH_0 + aG_0 - dH_1 + cG_1 + bH_2 - eG_2) \quad (\text{equ. 79})$$

As for the forward filtering process, the interleaved H and G data stream is multiplied by the relevant integer numerator and summed as shown. The output D data values are found by dividing the summations by the constant 64, which is also an integer power of 2.

To calculate the first and last H and G values, the filter equations must be altered such that values outside the boundaries of the data stream are not required. For example, if H_0 is to be calculated using the six coefficient filter, the values D_1 and D_2 would be required. Because

- 82 -

these values are not defined, a different filter is used at the beginning and end of the data stream. The new filters are determined such that the reconstruction process for the first and last two data values is possible. The following 5 pair of equations show the filter used to calculate the first H and G values:

$$H_0 = cD_0 - dD_1 - eD_2 + fD_3 \quad (\text{equ. 80})$$

$$G_0 = dD_0 + cD_1 - bD_2 + aD_3 \quad (\text{equ. 81})$$

The last H and G values are calculated with:

$$10 \quad H_3 = aD_1 + bD_2 + cD_3 - dD_4 \quad (\text{equ. 82})$$

$$G_3 = fD_1 - eD_2 + dD_3 + cD_4 \quad (\text{equ. 83})$$

In this case, these equations are equivalent to using the non-boundary equations with data values outside the data stream being equal to zero. The following inverse 15 transform boundary filters are used to reconstruct the first two and last two data values:

$$D_0 = 2 \left(\left(c - \frac{b}{\beta} \right) H_0 + \left(d + \frac{e}{\beta} \right) G_0 + aH_1 - fG_1 \right) \quad (\text{equ. 84})$$

$$D_1 = 2 \left(\left(\frac{a}{\beta} - d \right) H_0 + \left(c - \frac{f}{\beta} \right) G_0 + bH_1 - eG_1 \right) \quad (\text{equ. 85})$$

$$D_A = 2 \left(-eH_1 - bG_1 + \left(c - \frac{f}{\beta} \right) H_2 + \left(d - \frac{a}{\beta} \right) G_2 \right) \quad (\text{equ. 86})$$

$$D_B = 2 \left(fH_1 + aG_1 - \left(d + \frac{e}{\beta} \right) H_2 + \left(c - \frac{b}{\beta} \right) G_2 \right) \quad (\text{equ. 87})$$

INCREASING SOFTWARE DECOMPRESSION SPEED

A system is desired for compressing and decompressing video using dedicated digital hardware to compress and 20 using software to decompress. For example, in a video mail application one user uses a hardware compression expansion card for an IBM PC personal computer coupled to a video camera to record a video message in the form of a video message file. This compressed video message file is then 25 transmitted via electronic mail over a network such as a hardwired network of an office building. A recipient user receives the compressed video message file as he/she would receive a normal mail file and then uses the software to

- 83 -

decompress the compressed video message file to retrieve the video mail. The video mail may be displayed on the monitor of the recipient's personal computer. It is desirable to be able to decompress in software because
5 decompressing in software frees multiple recipients from purchasing relatively expensive hardware. Software for performing the decompression may, for example, be distributed free of charge to reduce the cost of the composite system.

10 In one prior art system, the Intel Indeo video compression system, a hardware compression expansion card compresses video and a software package is usable to decompress the compressed video. This system, however, only achieves a small compression ratio. Accordingly,
15 video picture quality will not be able to be improved as standard personal computers increase in computing power and/or video bandwidth.

The specification above discloses a method and apparatus for compressing and decompressing video. The
20 software decompression implementation written in the programming language C disclosed in Appendix A only decompresses at a few frames per second on a standard personal computer at the present date. A method capable of implementation in software which realizes faster
25 decompression is therefore desirable.

A method for decompressing video described above is therefore modified to increase software execution speed. Although the $b=19/32$, $a=11/32$, $c=5/32$ and $d=3/32$ coefficients used to realize the high and low pass forward
30 transform perfect reconstruction digital filters are used by dedicated hardware to compress in accordance with an above described method, the coefficients $b=5/8$, $a=3/8$, $c=1/8$ and $d=1/8$ are used to decompress in software on a digital computer. The coefficients are determined as shown
35 in the table below.

- 84 -

$$\begin{aligned}
 a &= \frac{1+\sqrt{3}}{8} = .3415(8) = 2.732 = \frac{3}{8} \\
 b &= \frac{3+\sqrt{3}}{8} = .5915(8) = 4.732 = \frac{5}{8} \\
 c &= \frac{3-\sqrt{3}}{8} = .1585(8) = 1.268 = \frac{1}{8} \\
 d &= \frac{-1+\sqrt{3}}{8} = .0915(8) = 0.732 = \frac{1}{8}
 \end{aligned}$$

5

Table 5

An even start inverse transform digital filter in accordance with the present embodiment is:

$$D_0 = 4[(b-a)H_0 + (c-d)G_0] \quad (\text{equ. 88})$$

where, for example, D_0 is a first inverse transformed data value indicative of a corresponding first data value of a row of the original image, and where H_0 and G_0 are first low and high pass component transformed data values of a row of a sub-band decomposition.

An odd end inverse transform digital filter in accordance with the present embodiment is:

$$D_1 = 4[(c+d)H_1 - (a+b)G_1] \quad (\text{equ. 89})$$

where, for example, D_1 is a last inverse transformed data value indicative of a corresponding last data value of a row of the original image, and where H_1 and G_1 are last low and high pass component transformed data values of a row of a sub-band decomposition.

An odd interleaved inverse transform digital filter in accordance with the present embodiment is:

$$\frac{D(2x-1)}{2} = \frac{1}{8}H(x-1) - \frac{5}{8}G(x-1) + \frac{3}{8}H(x) + \frac{1}{8}G(x) \quad (\text{equ. 90})$$

25 An even interleaved inverse transform digital filter in accordance with the present embodiment is:

$$\frac{D(2x)}{2} = -\frac{1}{8}H(x-1) + \frac{3}{8}G(x-1) + \frac{5}{8}H(x) + \frac{1}{8}G(x) \quad (\text{equ. 91})$$

As indicated by equations 90 and 91, the odd and even interleaved inverse transform digital filters operable on

- 85 -

the same H and G values of the sub-band decomposition but generate the odd and even inverse transformed data values in a row between the even start and odd end filters of equations 88 and 89. .

- 5 Using the above even start, odd end, odd interleaved and even interleaved inverse transform digital filters, a frame rate of approximately 15 frames/second is realizable executing on a Macintosh Quadra personal computer having a 68040 microprocessor. Digital filters using the
- 10 coefficients $b=5/8$, $a=3/8$, $c=1/8$ and $d=1/8$ may also be realized in dedicated digital hardware to reduce the cost of a dedicated hardware implementation where a slightly lower compression ratio is acceptable.

To further increase software decompression speed when

15 decompressing video on a digital computer, only two octaves of inverse transform are performed on video which was previously compressed using three octaves of forward transform. This results in the low pass component of the octave 0 decomposition. The low pass component of the

20 octave 0 decomposition is a non-aliased high quality quarter size decimated version of the original image. Rather than performing octave 0 of inverse transform, horizontal linear interpolation is used to expand each row of data values of the low pass component of the octave 0

25 decomposition into twice the number of data values. To expand the number of rows, each row of interpolated data values is replicated once so that the total number of rows is doubled. In some embodiments, interpolation techniques other than linear interpolation are used to improve image

30 quality. For example, spline interpolation or polynomial interpolation may be used.

To further increase software execution speed when decompressing video, luminance data values are decompressed using the digital filters of equations 88, 89, 90 and 91.

35 The chrominance data values, on the other hand, are decompressed using even and odd interleaved reconstruction filters having a fewer number of coefficients than four.

- 86 -

In one embodiment, two coefficient odd interleaved Haar and even interleaved Haar filters are used. The even interleaved Haar reconstruction filter is:

$$D_0 = (H_0 + G_0) \quad (\text{equ. 92})$$

5 The odd interleaved Haar reconstruction filter is:

$$D_1 = (H_0 - G_0) \quad (\text{equ. 93})$$

Because the above Haar filters each only have two coefficients, there is no boundary problem as is addressed in connection with an above-described method. Accordingly,
10 another start inverse transform digital filter and another end inverse transform digital filter are not used.

To increase software execution speed still further when decompressing video, variable-length SEND and STILL_SEND tokens are used. Data values are encoded using
15 a Huffman code as disclosed above whereas tokens are generated in variable-length form and appear in this variable-length form in the compressed data stream. This allows decompression to be performed without first calculating flags.

20 Figure 44 shows variable-length tokens used for encoding and decoding in accordance with some embodiments of the present invention. Because transitions from SEND mode to STOP mode or from STILL_SEND mode to STOP mode occur most frequently of the transitions indicated in
25 Figure 44, the corresponding tokens consist of only one bit.

In general, if an area changes from white to black in two consecutive frames of a video sequence and if the encoder is in LPF_SEND mode, then the difference between
30 the corresponding data values after quantization will be much larger than 37. 37 is the maximum number encodable using the specific Huffman code set forth in connection with an above-described method. Because such a large

- 87 -

change in data value cannot be encoded, an artifact will be generated in the decompressed image for any change in quantized data values exceeding 37. Accordingly, the Huffman code in the table below is used in accordance with one embodiment of the present invention.

	HUFFMAN CODE	qindex
	0	0
	1s1	± 1
	1s01	± 2
10	1s001	± 3
	1s0001	± 4
	1s00001	± 5
	1s000001	± 6
	1s0000001	± 7
15	1s0000000 ($ qindex - 8$)	$\pm 8 \dots \pm 135$

Table 6

In Table 6 above, the value ($|qindex| - 8$) is seven bits in length. The s in Table 6 above is a sign bit.

This embodiment is not limited to video mail applications and is not limited to systems using dedicated hardware to compress and software executing on a digital computer to decompress. Digital circuitry of a general purpose digital computer having a microprocessor may be used to decode and inverse transform a compressed image data stream. The coefficients $5/8$, $3/8$, $1/8$ and $1/8$ independent of sign may be the four coefficients of four coefficient high and low pass forward transform perfect reconstruction digital filters used to transform image data values into a sub-band decomposition.

- 88 -

Although the present invention has been described by way of the above described specific embodiments, it will be understood that certain adaptations, modifications, rearrangements and combinations of various features of the specific embodiments may be practiced without departing from the scope of the invention. Filters other than the four coefficient quasi-Daubechies filters can be used. In some embodiments, six coefficient quasi-Daubechies filters are used. Embodiments of this invention may, for example, be practiced using a one-dimensional tree structure, a two-dimensional tree structure, or a three-dimensional tree structure. Rather than testing whether or not a two-by-two block of data values is interesting, blocks of other sizes may be used. Three-by-three blocks of data values may, for example, be tested. Blocks of different sizes may be used in different octaves of a decomposition. In certain embodiments, there are different types of interesting blocks. The use of tokens in combination with use of a tree structure of a decomposition to reduce the number of data values encoded may be extended to include other tokens having other meanings. The "interesting with motion" token is but one example. Tree structures may be used in numerous ways to estimate the activity of a frame for rate control purposes. Numerous boundary filters, thresholds, encoder and decoder modes, token schemes, tree traversing address generators, quantization schemes, Huffman-like codes, and rate control schemes will be apparent from the specific embodiments. The above-described specific embodiments are therefore described for instructional purposes only and are not intended to limit the invention as set forth in the appended claims.

DATA COMPRESSION AND DECOMPRESSION
GREGORY KNOWLES AND ADRIAN S. LEWIS

M-2357 US

APPENDIX A

- 90 -

source/Bits.c

```
/*
    Reading and writing bits from a file
*/

#include    "../include/xwave.h"
#include    "../include/Bits.h"

Bits  bopen(name,mode)

String name, mode;

{
    Bits  bits = (Bits)MALLOC(sizeof(BitsRec));

    if((bits->fp = fopen(name,mode)) == (FILE*)0)Eprintf("Failed to open binary
file\n");    /*change*/
    bits->bufsize = 0;    /*new*/
    bits->buf = (unsigned char)0;    /*new*/
    return(bits);
}

void  bclose(bits)

Bits  bits;

{
    if(fclose(bits->fp)!=0) Eprintf("Failed to close binary file\n"); /*was:
fclose(bits->fp)*/
```

- 91 -

```

        XtFree(bits);
    }

void bread(bytes,num,bits)

unsigned char    *bytes;
int    num;
Bits    bits;

{
    int    byte=0, bit=0,pull,b;

    bytes[byte]=0;
    while(num>0) {
        if (bits->bufsize==0) {
            pull=fgetc(bits->fp);
            if(pull==EOF)
            {
                /*printf("EOF\n");  Previously didn't check for
EOF:bits->buf=(unsigned char)fgetc(bits->fp)*/
                for(b=byte+1;b<num/8+1;b++)
                    bytes[b]=(unsigned char)0;
                return;
            }
            bits->buf=(unsigned char)pull;
            bits->bufsize=8;
        }

        bytes[byte]=((1&bits->buf)!=0)?bytes[byte]|(1<<bit):bytes[byte]&~(1<<bit);
        if (bit==7) { bit=0; byte++; bytes[byte]=0; } /* was bit==8 */
        else bit++;
        bits->buf=bits->buf>>1;
    }
}

```

- 92 -

```
        bits->bufsize--;  
        num--;  
    }  
}  
  
void bwrite(bytes,num,bits)  
  
unsigned char    *bytes;  
int    num;  
Bits    bits;  
  
{  
    int    byte=0, bit=0;  
    unsigned char    xfer;  
  
    while(num > 0) {  
        if (bit==0) {  
            xfer=bytes[byte++];
```

- 93 -

source/Color.c

/*

* Color routines

*/

#include "../include/xwave.h"

#define GAMMA 1.0/2.2

int

VisualClass[6] = {PseudoColor, DirectColor, TrueColor, StaticColor, GrayScale, StaticGray};

/* Function Name: Range

* Description: Range convert for RGB/YUV calculations

* Arguments: old_x - old value (0..old_r-1)

* old_r - old range < new_r

* new_r - new range

* Returns: old_x scaled up to new range

*/

int Range(old_x, old_r, new_r)

int old_x, old_r, new_r;

{

return((old_x*new_r)/old_r);

}

/* Function Name: Gamma

* Description: Range convert with Gamma correction for RGB/YUV calculations

* Arguments: as Range +

* factor - gamma correction factor

- 94 -

* Returns: old_x gamma corrected and scaled up to new range

*/

```
int Gamma(old_x,old_r,new_r,factor)
```

```
int old_x, old_r, new_r;
```

```
double factor;
```

```
{
    return((int)((double)new_r*pow((double)old_x/(double)old_r,factor)));
}
```

/* Function Name: Dither

* Description: Range convert with dithering for RGB/YUV calculations

* Arguments: levels - output range (0..levels-1)

* pixel - pixel value ($0 \leq \text{pixel} < 8 + \text{precision}$)

* x, y - dither location

* precision - pixel range ($0 \leq \text{precision} < 8 + \text{precision}$)

* Returns: dithered value (0..levels-1)

*/

```
int Dither(levels,pixel,x,y,precision)
```

```
int pixel, levels, x, y, precision;
```

```
{
    int bits=8+precision,
        pixlev=pixel*levels,
```

```
value=(pixlev >> bits)+((pixlev-(pixlev&(-1 << bits)))>> precision> global->dither{x
&15}[y&15]?1:0);
```

- 95 -

```
    return(value >= levels?levels-1:value);
}

/*  Function Name:    ColCvt
 *   Description:    Converts between RGB and YUV triples
 *   Arguments:    src - source triple
 *                  dst - destination triple
 *                  rgb_yuv - convert direction RGB->YUV True
 *                  max - range of data (max-1..-max)
 *   Returns:      alters dst.
 */

void ColCvt(src,dst,rgb_yuv,max)

short src[3], dst[3];
Boolean rgb_yuv;
int max;

{
    double rgb_yuv_mat[2][3][3]={
        {0.299,0.587,0.114},
        {-0.169,-0.3316,0.5},
        {0.5,-0.4186,-0.0813}
    },{
        {1,0,1.4021},
        {1,-0.3441,-0.7142},
        {1,1.7718,0}
    }
};
    int i, channel;

    for(channel=0;channel<3;channel++) {
```

- 96 -

```

double      sum=0.0;

    for(i=0;i<3;i++)
sum +=(double)(src[i])*rgb_yuv_mat[rgb_yuv?0:1][channel][i];
    dst[channel]=(int)sum < -max?-max:(int)sum > max-1?max-1:(short)sum;
    }
}

/*  Function Name:      CompositePixel
*   Description:  Calculates pixel value from components
*   Arguments:   frame - Frame to be drawn on
*                x, y - coordinate of pixel in data
*                X, Y - coordinate of pixel in display
*   Returns:     pixel value in colormap
*/

int  CompositePixel(frame,x,y,X,Y)

Frame frame;
int   x, y, X, Y;

{
    Video vid=frame->video;
    int   channel=frame->channel, pixel, value=0;

    if (channel!=3) {

pixel=(int)vid->data[channel][frame->frame][Address2(vid,channel,x,y)]+(128<<vid-
>precision);
        value=Dither(global->levels,pixel,X,Y,vid->precision);
    } else for(channel=0;channel<3;channel++) {
        int

```

- 97 -

```
levels=vid->type==RGB?global->rgb_levels:global->yuv_levels[channel];
```

```
pixel=(int)vid->data[channel][frame->frame][Address(vid,channel,x,y)]+(128<<vid->precision),
```

```
value=levels*value+Dither(levels,pixel,X,Y,vid->precision);
```

```
}
```

```
return(value);
```

```
}
```

```
void InitVisual()
```

```
{
```

```
Display *dpy=XtDisplay(global->toplevel);
```

```
int scrn=XDefaultScreen(dpy), class=0, depth=8, map, i, r, g, b, y, u, v;
```

```
String
```

```
VisualNames[6]={"PseudoColor","DirectColor","TrueColor","StaticColor","GrayScale",  
"StaticGray"};
```

```
XColor color;
```

```
global->visinfo=(XVisualInfo *)MALLOC(sizeof(XVisualInfo));
```

```
while(depth>0
```

```
&&!XMatchVisualInfo(dpy,scrn,depth,VisualClass[class],global->visinfo))
```

```
if (class==5) {class=0; depth--;} else class++;
```

```
Dprintf("Visual: %s depth %d\n",VisualNames[class],depth);
```

```
global->palettes=(Palene)MALLOC(sizeof(PaleneRec));
```

```
strcpy(global->palettes->name,"Normal");
```

```
global->palettes->next=NULL;
```

```
global->no_pals=1;
```

```
switch(global->visinfo->class) {
```

```
case TrueColor:
```

```
case DirectColor:
```

- 98 -

```

case StaticColor:
case GrayScale:
    fprintf(stderr, "Unsupported visual type: %s\n", VisualNames[class]);
    exit();
    break;
case PseudoColor:
    global->levels = global->visinfo->colormap_size;
    global->rgb_levels = (int)pow((double)global->levels, 1.0/3.0);
    for(map=0; map<2; map++) { /* rgb non-gamma and gamma maps */

global->cmaps[map] = XCreateColormap(dpy, XDefaultRootWindow(dpy), global->visinfo
->visual, AllocAll);

        for(r=0; r<global->rgb_levels; r++)
            for(g=0; g<global->rgb_levels; g++)
                for(b=0; b<global->rgb_levels; b++) {

color.pixel = (r*global->rgb_levels + g)*global->rgb_levels + b;

color.red = (map&1)?Gamma(r, global->rgb_levels, 65536, GAMMA):Range(r, global->rg
b_levels, 65536);

color.green = (map&1)?Gamma(g, global->rgb_levels, 65536, GAMMA):Range(g, global->
rgb_levels, 65536);

color.blue = (map&1)?Gamma(b, global->rgb_levels, 65536, GAMMA):Range(b, global->r
gb_levels, 65536);

color.flags = DoRed | DoGreen | DoBlue;

XStoreColor(dpy, global->cmaps[map], &color);

        }

color.pixel = global->levels-1;
color.red = 255 < < 8;

```

- 99 -

```

        color.green=255 < < 8;
        color.blue=255 < < 8;
        color.flags=DoRed | DoGreen | DoBlue;
        XStoreColor(dpy,global->cmaps[map],&color);
    }
    for(map=2;map<4;map++) { /* mono non-gamma and gamma maps */

global->cmaps[map]=XCreateColormap(dpy,XDefaultRootWindow(dpy),global->visinfo
->visual,AllocAll);

        for(i=0;i<global->visinfo->colormap_size;i++) {
            color.pixel=i;

color.red=(map&1)?Gamma(i,global->levels,65536,GAMMA):Range(i,global->levels,6
5536);

color.green=(map&1)?Gamma(i,global->levels,65536,GAMMA):Range(i,global->levels
,65536);

color.blue=(map&1)?Gamma(i,global->levels,65536,GAMMA):Range(i,global->levels,
65536);

            color.flags=DoRed | DoGreen | DoBlue;
            XStoreColor(dpy,global->cmaps[map],&color);
        }
    }
    global->yuv_levels[0]=(int)pow((double)global->levels,1.0/2.0);
    global->yuv_levels[1]=(int)pow((double)global->levels,1.0/4.0);
    global->yuv_levels[2]=(int)pow((double)global->levels,1.0/4.0);
    for(map=4;map<6;map++) { /* yuv non-gamma and gamma maps */

global->cmaps[map]=XCreateColormap(dpy,XDefaultRootWindow(dpy),global->visinfo
->visual,AllocAll);

        for(y=0;y<global->yuv_levels[0];y++)

```

- 100 -

```

        for(u=0;u < global->yuv_levels[1];u++)
            for(v=0;v < global->yuv_levels[2];v++) {
                short
src[3] = {(short)(Range(y,global->yuv_levels[0],65536)-32768),
(short)(Range(u,global->yuv_levels[1],65536)-32768),
(short)(Range(v,global->yuv_levels[2],65536)-32768)}, dst[3];

                ColCvt(src,dst,False,65536/2);

color.pixel=(y*global->yuv_levels[1]+u)*global->yuv_levels[2]+v;

color.red=(map&1)?Gamma((int)dst[0]+32768,65536,65536,GAMMA):(int)dst[0]+32768;

color.green=(map&1)?Gamma((int)dst[1]+32768,65536,65536,GAMMA):(int)dst[1]+32768;

color.blue=(map&1)?Gamma((int)dst[2]+32768,65536,65536,GAMMA):(int)dst[2]+32768;

                color.flags=DoRed | DoGreen | DoBlue;

XStoreColor(dpy,global->cmaps[map],&color);
            }
        color.pixel=global->levels-1;
        color.red=255 < < 8;
        color.green=255 < < 8;
        color.blue=255 < < 8;
        color.flags=DoRed | DoGreen | DoBlue;
        XStoreColor(dpy,global->cmaps[map],&color);
    }

```

- 101 -

```

    global->palettes->mappings=NULL;
    break;
case StaticGray:
    global->levels=1<<depth;
    for(i=0;i<6;i++) global->cmaps[i]=XDefaultColormap(dpy,scrn);
    color.pixel=0;
    XQueryColor(dpy,XDefaultColormap(dpy,scrn),&color);
    if (color.red==0 && color.green==0 && color.blue==0)
global->palettes->mappings=NULL;
        else {
            global->palettes->mappings=(Map)MALLOC(sizeof(MapRec));
            global->palettes->mappings->start=0;
            global->palettes->mappings->finish=global->levels-1;
            global->palettes->mappings->m=-1;
            global->palettes->mappings->c=global->levels-1;
            global->palettes->mappings->next=NULL;
        }
    break;
}
}

```

```

Colormap    ChannelCmap(channel,type,gamma)

```

```

int    channel;

```

```

VideoFormat type;

```

```

Boolean    gamma;

```

```

{

```

```

    Colormap    cmap;

```

```

    if (channel!=3 || type==MONO) {

```

```

        if (gamma) cmap=global->cmaps[global->cmaps[2]==NULL?3:2];
    }
}

```


- 102 -

```
        else cmap=global->cmaps[global->cmaps[3] == NULL?2:3];  
    } else if (type == RGB) {  
        if (gamma) cmap=global->cmaps[global->cmaps[0] == NULL?1:0];  
        else cmap=global->cmaps[global->cmaps[1] == NULL?0:1];  
    } else {  
        if (gamma) cmap=global->cmaps[global->cmaps[4] == NULL?5:4];  
        else cmap=global->cmaps[global->cmaps[5] == NULL?4:5];  
    }  
    return(cmap);  
}
```

- 103 -

source/Convert.c

```
#include    "../include/xwave.h"

short  cti(c)

char  c;

{
    return((short)(c)^-128);
}

char  itc(i)

short  i;

{
    static int    errors=0;
    if (i<-128 || i>127) {
        if (errors == 99) {
            Dprintf("100 Conversion overflows\n");
            errors=0;
        } else errors++;
        i=(i<-128)?-128:127;
    }
    return((char)(i^128));
}
```

- 104 -

source/Convolve3.c

```
/*
    2D wavelet transform convolver (fast hardware emulation)
    New improved wavelet coeffs : 11 19 5 3
*/

#include    "../include/xwave.h"

/*    Function Name:    Round
 *    Description:    Rounding to a fixed number of bits, magnitude rounded down
 *    Arguments:    number - number to be rounded
 *                  bits - shifted bits lost from number
 *    Returns:    rounded number
 */

short Round(number,bits)

int    number;
int    bits;

{
    if (bits == 0) return((short)number);
    else return((short)(number + (1 << bits-1) - (number < 0 ? 0 : 1) >> bits));
}

/*    Function Name:    Convolve
 *    Description:    Perform a wavelet convolution on image data
 *    Arguments:    data - data to be transformed
 *                  dirn - convolution direction
```

- 105 -

```

*           size - size of image data
*           oct_src, oct_dst - initial and final octave numbers
*   Returns:   data altered
*/

```

```
void Convolve(data,dirn,size,oct_src,oct_dst)
```

```
short *data;
```

```
Boolean   dirn;
```

```
int   size[2], oct_src, oct_dst;
```

```
{
```

```
    int   tab[4][4], addr[4]={-1,-1,-1,-1}, index, mode, i, j, oct, orient,
area=size[0]*size[1];
```

```
    Boolean   fwd_rev=oct_src<oct_dst;
```

```
    int   windows[12][5]={
        {1,2,3,-4,2}, /* 0 - normal forward 0 */
        {4,-3,2,1,3}, /* 1 - normal forward 1 */
        {1,-2,3,4,2}, /* 2 - normal reverse 0 */
        {4,3,2,-1,3}, /* 3 - normal reverse 1 */
        {2,3,4,-4,3}, /* 4 - end forward 0 */
        {4,-4,3,2,4}, /* 5 - end forward 1 */
        {2,2,3,-4,2}, /* 6 - start forward 0 */
        {4,-3,2,2,3}, /* 7 - start forward 1 */
        {3,-4,-4,3,4}, /* 8 - break reverse end dirn==False*/
        {4,3,-3,-4,3}, /* 9 - break reverse start dirn==False */
        {-3,-4,4,3,4}, /* 10 - break reverse end dirn==True */
        {-4,3,3,-4,3}, /* 11 - break reverse start dirn==True */
    }, win[3];           /* 12 - no calculation */
```

```
    for(oct=oct_src;oct!=oct_dst;oct+=(fwd_rev?1:-1)) {
```

```
        long   shift=oct-(fwd_rev?0:1);
```

- 106 -

```

for(orient=0;orient<2;orient++) {
    Boolean    x_y=fwd_rev==(orient==0);

for (index=0;index<(area>>(shift<<1));index++) {
    long  major, minor, value, valuex3, valuex11, valuex19, valuex5;

    major=index/(size[x_y?0:1]>>shift);
    minor=index-major*(size[x_y?0:1]>>shift);
    for(j=0;j<3;j++) win[j]=12;
    switch(minor) {
    case 0: break;
    case 1: if (!fwd_rev) win[0]=dirn?11:9; break;
    case 2: if (fwd_rev) { win[0]=6; win[1]=7; }; break;
    default:
        if (minor+1==size[x_y?0:1]>>shift) {
            if (fwd_rev) { win[0]=4; win[1]=5; }
            else { win[0]=2; win[1]=3; win[2]=dirn?10:8; }
        } else if (fwd_rev) {
            if ((1&minor)==0) { win[0]=0; win[1]=1; }
        } else {
            if ((1&minor)!=0) { win[0]=2; win[1]=3; }
        }
    }

    addr[3&index]=(x_y?minor:major)+size[0]*(x_y?major:minor)<<shift;
    value=(int)data[addr[3&index]];

    valuex5=value+(value<<2);
    valuex3=value+(value<<1);
    valuex11=valuex3+(value<<3);
    valuex19=valuex3+(value<<4);
    tab[3&index][3]=fwd_rev || !dirn?valuex3:valuex19;
    tab[3&index][2]=fwd_rev || dirn?valuex5:valuex11;

```

- 107 -

```
tab[3&index][1]=fwd_rev || !dirn?valuex19:valuex3;
tab[3&index][0]=fwd_rev || dirn?valuex11:valuex5;
for(j=0;j<3 && win[j]!=12;j++) {
    int    conv=0;

    for(i=0;i<4;i++) {
        int    wave=dirn?3-i:i;

conv += negif(0> windows[win[j]][wave],tab[3&index+abs(windows[win[j]]][i])[wave]);
    }

data[addr[3&index+ windows[win[j]][4]]]=Round(conv,fwd_rev?5:win[j]>7?3:4);
    }
}}}
}
```

source/Copy.c

```
/*
    Copy video, includes direct copy, differencing, LPF zero, LPF only, RGB-YUV
    conversion and gamma correction
*/

#include    "../include/xwave.h"
#include    "Copy.h"
extern int  Shift();
extern void ColCvt();

void  CopyVideoCtrl(w,closure,call_data)

Widget      w;
caddr_t closure, call_data;

{
    CopyCtrl  ctrl=(CopyCtrl)closure;
    Video  new=CopyHeader(ctrl->video), src=ctrl->video;
    int    frame, channel, i, x, y, X, Y, map[256];

    if (global->batch == NULL)
ctrl->mode=(int)XawToggleGetCurrent(ctrl->radioGroup);
    strcpy(new->name,ctrl->name);
    strcpy(new->files,new->name);
    switch(ctrl->mode) {
case 1:    Dprintf("Direct copy\n");
            new->UVsample[0]=ctrl->UVsample[0];
            new->UVsample[1]=ctrl->UVsample[1];
```

- 109 -

```

        break;
case 2:    Dprintf("Differences\n");
        break;
case 3:    Dprintf("LPF zero\n");
        break;
case 4:    Dprintf("LPF only\n");
        new->trans.type = TRANS_None;

new->size[0] = new->size[0] >> new->trans.wavelet.space[0];

new->size[1] = new->size[1] >> new->trans.wavelet.space[0];
        break;
case 5:    Dprintf("RGB-YUV\n");
        new->type = new->type == YUV?RGB:YUV;
        new->UVsample[0] = 0;
        new->UVsample[1] = 0;
        break;
case 6:    Dprintf("Gamma conversion\n");
        new->gamma = !new->gamma;
        for(i=0; i<256; i++)
map[i] = gamma(i,256,new->gamma?0.5:2.0);
        break;
}
if (new->disk == True) SaveHeader(new);
for(frame=0; frame<new->size[2]; frame++) {
    GetFrame(src, frame);
    NewFrame(new, frame);
    switch(ctrl->mode) {
case 1:
for(channel=0; channel<(new->type == MONO?1:3); channel++) {
        int    size = Size(new, channel, 0)*Size(new, channel, 1);

```


- 110 -

```
for(y=0;y < Size(new,channel,1);y++)
```

```
for(x=0;x < Size(new,channel,0);x++)
```

```
new->data[channel][frame][x+Size(new,channel,0)*y]=src->data[channel][frame][Shift(
```

```
x,src->type == YUV &&
```

```
channel!=0?new->UVsample[0]-src->UVsample[0]:0)+Size(src,channel,0)*Shift(y,src-
```

```
>type == YUV && channel!=0?new->UVsample[1]-src->UVsample[1]:0]);
```

```
}
```

```
break;
```

```
case 2:
```

```
for(channel=0;channel < (new->type == MONO?1:3);channel++) {
```

```
int
```

```
size = Size(new,channel,0)*Size(new,channel,1);
```

```
for(i=0;i < size;i++)
```

```
new->data[channel][frame][i]=src->data[channel][frame][i]-(frame == 0?0:src->data[ch  
annel][frame-1][i]);
```

```
}
```

```
break;
```

```
case 3:
```

```
for(channel=0;channel < (new->type == MONO?1:3);channel++) {
```

```
int
```

```
size = Size(new,channel,0)*Size(new,channel,1);
```

```
for(i=0;i < size;i++) {
```

```
x=i%Size(new,channel,0);
```

```
y=i/Size(new,channel,0);
```

```
if
```

```
(x%(1 < < new->trans.wavelet.space[new->type == YUV && channel!=0?1:0]) == 0
```

```
&& y%(1 < < new->trans.wavelet.space[new->type == YUV &&
```

```
channel!=0?1:0]) == 0)
```

- 111 -

```

new->data[channel][frame][i]=0;

                                else
new->data[channel][frame][i]=src->data[channel][frame][i];
                                }
                                }
                                break;

        case 4:
for(channel=0;channel<(new->type==MONO?1:3);channel++) {
                                int
size=Size(new,channel,0)*Size(new,channel,1);

                                for(i=0;i<size;i++) {
                                        x=i%Size(new,channel,0);
y=i/Size(new,channel,0);

new->data[channel][frame][i]=src->data[channel][frame][(x+(y<<new->trans.wavele
t.space[0])*Size(new,channel,0))<<new->trans.wavelet.space[0]];
                                }
                                }
                                break;

        case 5:    for(X=0;X<new->size[0];X++)
for(Y=0;Y<new->size[1];Y++) {

                                short  src_triple[3], dst_triple[3];

                                for(channel=0;channel<3;channel++)

src_triple[channel]=src->data[channel][frame][Address(src,channel,X,Y)];

ColCvt(src_triple,dst_triple,new->type==YUV,1<<7+new->precision);

                                for(channel=0;channel<3;channel++)
                                        new->data[channel][frame][Address(new,channel,X,Y)]=dst_triple[channel];
                                }

```

- 112 -

```

        break;

    case 6:
        for(channel=0;channel < (new->type == MONO?1:3);channel++) {
            int
            size = Size(new,channel,0)*Size(new,channel,1);

            for(i=0;i < size;i++)
                new->data[channel][frame][i] = map[src->data[channel][frame][i] + 128]-128;
            }
            break;
        }

        if (frame > 0) FreeFrame(src,frame-1);
        SaveFrame(new,frame);
        FreeFrame(new,frame);
    }

    FreeFrame(src,src->size[2]-1);
    new->next = global->videos;
    global->videos = new;
}

void BatchCopyCtrl(w,closure,call_data)

Widget      w;
caddr_t     closure, call_data;

{
    CopyCtrl    ctrl=(CopyCtrl)closure;

    if (ctrl->video == NULL)
        ctrl->video = FindVideo(ctrl->src_name,global->videos);
    CopyVideoCtrl(w,closure,call_data);
}

```

- 113 -

```
CopyCtrl    InitCopyCtrl(name)
```

```
String name;
```

```
{
    CopyCtrl    ctrl=(CopyCtrl)MALLOC(sizeof(CopyCtrlRec));

    strcpy(ctrl->src_name,name);
    strcpy(ctrl->name,name);
    ctrl->mode=1;
    return(ctrl);
}
```

```
#define      COPY_ICONS      17
```

```
void CopyVideo(w,closure,call_data)
```

```
Widget      w;
```

```
caddr_t      closure, call_data;
```

```
{
    Video video=(Video)closure;
    CopyCtrl    ctrl=InitCopyCtrl(video->name);
    NumInput    UVinputs=(NumInput)MALLOC(2*sizeof(NumInputRec));
    Message      msg=NewMessage(ctrl->name,NAME_LEN);
    XtCallbackRec    destroy_call[]={
        {Free,(caddr_t)ctrl},
        {Free,(caddr_t)UVinputs},
        {CloseMessage,(caddr_t)msg},
        {NULL,NULL},
    };
    Widget      shell=ShellWidget("copy_video",w,SW_below,NULL,destroy_call),
```

- 114 -

```

        form = FormatWidget("cpy_form".shell), widgets[COPY_ICONS];
FormItem  items[] = {
    {"cpy_cancel", "cancel", 0, 0, FW_icon, NULL},
    {"cpy_confirm", "confirm", 1, 0, FW_icon, NULL},
    {"cpy_title", "Copy a video", 2, 0, FW_label, NULL},
    {"cpy_vid_lab", "Video Name:", 0, 3, FW_label, NULL},
    {"cpy_text", NULL, 4, 3, FW_text, (String)msg},

    {"cpy_copy", "copy", 0, 5, FW_toggle, NULL},
    {"cpy_diff", "diff", 6, 5, FW_toggle, (String)6},
    {"cpy_lpf_zero", "lpf_zero", 7, 5, FW_toggle, (String)7},
    {"cpy_lpf_only", "lpf_only", 8, 5, FW_toggle, (String)8},
    {"cpy_color", "color_space", 9, 5, FW_toggle, (String)9},

    {"cpy_gamma", "gamma", 10, 5, FW_toggle, (String)10},
    {"cpy_UV0_int", NULL, 0, 6, FW_integer, (String)&UVinputs[0]},
    {"cpy_UV0_down", NULL, 12, 6, FW_down, (String)&UVinputs[0]},
    {"cpy_UV0_up", NULL, 13, 6, FW_up, (String)&UVinputs[0]},
    {"cpy_UV1_int", NULL, 0, 14, FW_integer, (String)&UVinputs[1]},

    {"cpy_UV1_down", NULL, 12, 14, FW_down, (String)&UVinputs[1]},
    {"cpy_UV1_up", NULL, 16, 14, FW_up, (String)&UVinputs[1]},
};

XtCallbackRec  callbacks[] = {
    {Destroy, (caddr_t)shell},
    {NULL, NULL},
    {CopyVideoCtrl, (caddr_t)ctrl},
    {Destroy, (caddr_t)shell},
    {NULL, NULL},
    {NULL, NULL}, {NULL, NULL}, {NULL, NULL}, {NULL, NULL},
{NULL, NULL}, {NULL, NULL},
    {NumIncDec, (caddr_t)&UVinputs[0]}, {NULL, NULL},

```

- 115 -

```

        {NumIncDec,(caddr_t)&UVinputs[0]}, {NULL,NULL},
        {NumIncDec,(caddr_t)&UVinputs[1]}, {NULL,NULL},
        {NumIncDec,(caddr_t)&UVinputs[1]}, {NULL,NULL},
    };

    Dprintf("CopyVideo\n");

    msg->rows=1; msg->cols=NAME_LEN;
    ctrl->video=video;
    UVinputs[0].format="UV sub-sample X: %d";
    UVinputs[0].min=0;
    UVinputs[0].max=2;
    UVinputs[0].value = &ctrl->UVsample[0];
    UVinputs[1].format="UV sub-sample Y: %d";
    UVinputs[1].min=0;
    UVinputs[1].max=2;
    UVinputs[1].value = &ctrl->UVsample[1];

    ctrl->UVsample[0]=video->UVsample[0];
    ctrl->UVsample[1]=video->UVsample[1];
    FillForm(form,COPY_ICONS,items,widgets,callbacks);
    ctrl->radioGroup=widgets[5];
    XtSetSensitive(widgets[6],video->size[2]>1);
    XtSetSensitive(widgets[7],video->trans.type!=TRANS_None);
    XtSetSensitive(widgets[8],video->trans.type!=TRANS_None);
    XtSetSensitive(widgets[9],video->type!=MONO);
    XtSetSensitive(widgets[10],video->type!=YUV &&
video->trans.type==TRANS_None);
    XtPopup(shell,XtGrabExclusive);
};

```

- 116 -

source/Frame.c

```
/*
    Frame callback routines for Destroy
*/

#include    "../include/xwave.h"
#include    <X11/Xmu/SysUtil.h>
#include    <pwd.h>
extern void CvtIndex();
extern Palette FindPalette();
extern void SetSensitive();

typedef    struct {
    Frame frame;
    int    frame_number, frame_zoom, frame_palette, frame_channel;
} ExamCtrlRec, *ExamCtrl;

void    FrameDestroy(w, closure, call_data)

Widget    w;
caddr_t    closure, call_data;

{
    Frame frame=(Frame)closure;
    void    CleanUpPoints(), FrameDelete();

    Dprintf("FrameDestroy\n");
    frame->point->usage--;
    if (frame->msg!=NULL) {
```

- 117 -

```
        frame->msg->shell=NULL;
        CloseMessage(NULL,(caddr_t)frame->msg,NULL);
    }
    if (frame->point->usage==0) CleanUpPoints(&global->points);
    XtPopdown(frame->shell);
    XtDestroyWidget(frame->shell);
    FrameDelete(&global->frames,frame);
}

void CleanUpPoints(points)

Point *points;

{
    Point dummy=*points;

    if (dummy!=NULL) {
        if (dummy->usage<1) {
            *points=dummy->next;
            XtFree(dummy);
            CleanUpPoints(points);
        } else CleanUpPoints(&((*points)->next));
    };
}

void FrameDelete(frames,frame)

Frame *frames, frame;

{
    if (*frames!=NULL) {
        if (*frames==frame) {
```


- 118 -

```

        int    number = frame->frame;

        frame->frame = -1;
        FreeFrame(frame->video,number);
        *frames = frame->next;
        XtFree(frame);
    } else FrameDelete(&(*frames)->next.frame);
}

}

void  ExamineCtrl(w,closure,call_data)

Widget      w;
caddr_t     closure, call_data;

{
    ExamCtrl  ctrl = (ExamCtrl)closure;
    Arg       args[1];

    if (ctrl->frame->frame != ctrl->frame_number-ctrl->frame->video->start) {
        int    old_frame = ctrl->frame->frame;

        ctrl->frame->frame = ctrl->frame_number-ctrl->frame->video->start;
        FreeFrame(ctrl->frame->video,old_frame);
        GetFrame(ctrl->frame->video,ctrl->frame->frame);
    }

    ctrl->frame->zoom = ctrl->frame_zoom;
    ctrl->frame->palette = ctrl->frame_palette;
    ctrl->frame->channel = ctrl->frame_channel;
    XtSetArg(args[0],XtNbixmap,UpdateImage(ctrl->frame));
    XtSetValues(ctrl->frame->image_widget,args,ONE);

```

- 119 -

```

XtSetArg(args[0],XtNcolormap,ChannelCmap(ctrl->frame->channel,ctrl->frame->vide
o->type,ctrl->frame->video->gamma));
    XtSetValues(ctrl->frame->shell,args,ONE);
    if (ctrl->frame->msg!=NULL) UpdateInfo(ctrl->frame);
}

```

```

#define      EXAM_ICONS      13

```

```

void  Examine(w,closure,call_data)

```

```

Widget      w;

```

```

caddr_t      closure, call_data;

```

```

{
    ExamCtrl  ctrl=(ExamCtrl)MALLOC(sizeof(ExamCtrlRec));
    NumInput  num_inputs=(NumInput)MALLOC(2*sizeof(NumInputRec));
    XtCallbackRec destroy_call[]={
        {Free,(caddr_t)ctrl},
        {Free,(caddr_t)num_inputs},
        {NULL,NULL},
    }, pal_call[2*global->no_pals];
    Widget      shell=ShellWidget("examine",w,SW_below,NULL,destroy_call),
               form=FormatWidget("exam_form",shell), widgets[EXAM_ICONS],
               pal_widgets[global->no_pals], pal_shell;
    Frame frame=(Frame)closure;
    FormItem  items[]={
        {"exam_cancel","cancel",0,0,FW_icon,NULL},
        {"exam_confirm","confirm",1,0,FW_icon,NULL},
        {"exam_label","Examine",2,0,FW_label,NULL},
        {"exam_ch_lab","Channel :",0,3,FW_label,NULL},

        {"exam_ch_btn",ChannelName[frame->video->type][frame->channel],4,3,FW_button,"

```

- 120 -

```

exam_cng_ch"},
    {"exam_pal_lab", "Palette : ", 0, 4, FW_label, NULL},

{"exam_pal_btn", FindPalette(global->palettes, frame->palette)->name, 4, 4, FW_button, "
exam_cng_pal"},
    {"exam_z_int", NULL, 0, 6, FW_integer, (String)&num_inputs[0]},
    {"exam_z_down", NULL, 8, 6, FW_down, (String)&num_inputs[0]},
    {"exam_z_up", NULL, 9, 6, FW_up, (String)&num_inputs[0]},
    {"exam_zoom_int", NULL, 0, 8, FW_integer, (String)&num_inputs[1]},
    {"exam_zoom_down", NULL, 8, 8, FW_down, (String)&num_inputs[1]},
    {"exam_zoom_up", NULL, 12, 8, FW_up, (String)&num_inputs[1]},
};

MenuItem    pal_menu[global->no_pals];
XtCallbackRec    callbacks[] = {
    {Destroy, (caddr_t)shell},
    {NULL, NULL},
    {ExamineCtrl, (caddr_t)ctrl},
    {Destroy, (caddr_t)shell},
    {NULL, NULL},
    {NumIncDec, (caddr_t)&num_inputs[0]}, {NULL, NULL},
    {NumIncDec, (caddr_t)&num_inputs[0]}, {NULL, NULL},
    {NumIncDec, (caddr_t)&num_inputs[1]}, {NULL, NULL},
    {NumIncDec, (caddr_t)&num_inputs[1]}, {NULL, NULL},
};

int    i, width=0;
Palette    pal=global->palettes;
XFontStruct    *font;
Arg    args[1];
caddr_t    dummy[global->no_pals], dummy2[global->no_pals]; /*
gcc-mc68020 bug avoidance */

Dprintf("Examine\n");

```

- 121 -

```
ctrl->frame = frame;
ctrl->frame_number = frame->frame + frame->video->start;
ctrl->frame_zoom = frame->zoom;
ctrl->frame_palette = frame->palette;
ctrl->frame_channel = frame->channel;
num_inputs[0].format = "Frame: %03d";
num_inputs[0].max = frame->video->start + frame->video->size[2]-1;
num_inputs[0].min = frame->video->start;
num_inputs[0].value = &ctrl->frame_number;
num_inputs[1].format = "Zoom: %d";
num_inputs[1].max = 4;
num_inputs[1].min = 0;
num_inputs[1].value = &ctrl->frame_zoom;
```

```
FillForm(form, EXAM_ICONS, items, widgets, callbacks);
```

```
font = FindFont(widgets[6]);
for(i=0; pal!=NULL; pal=pal->next, i++) {
    pal_menu[i].name = pal->name;
    pal_menu[i].widgetClass = smeBSBObjectClass;
    pal_menu[i].label = pal->name;
    pal_menu[i].hook = NULL;
    pal_call[i*2].callback = SimpleMenu;
    pal_call[i*2].closure = (caddr_t)&ctrl->frame_palette;
    pal_call[i*2+1].callback = NULL;
    pal_call[i*2+1].closure = NULL;
    width = TextWidth(width, pal->name, font);
}
pal_shell = ShellWidget("exam_cng_pal", shell, SW_menu, NULL, NULL);
FillMenu(pal_shell, global->no_pals, pal_menu, pal_widgets, pal_call);
XtSetArg(args[0], XtNwidth, 2 + width);
XtSetValues(widgets[6], args, ONE);
```

- 122 -

```

if (frame->video->type == MONO) XtSetSensitive(widgets[4],False);
else {
    MenuItem    ch_menu[4];
    Widget
ch_shell=ShellWidget("exam_cng_ch",shell,SW_menu,NULL,NULL), ch_widgets[4];
    XtCallbackRec    ch_call[8];

    font=FindFont(widgets[4]);
    width=0;
    for(i=0;i<4;i++) {
        ch_menu[i].name=ChannelName[frame->video->type][i];
        ch_menu[i].widgetClass=smeBSBObjectClass;
        ch_menu[i].label=ChannelName[frame->video->type][i];
        ch_menu[i].hook=(caddr_t)&ctrl->frame_channel;
        ch_call[i*2].callback=SimpleMenu;
        ch_call[i*2].closure=(caddr_t)&ctrl->frame_channel;
        ch_call[i*2+1].callback=NULL;
        ch_call[i*2+1].closure=NULL;

        width=TextWidth(width,ChannelName[frame->video->type][i],font);
    }
    FillMenu(ch_shell,4,ch_menu,ch_widgets,ch_call);
    XtSetArg(args[0],XtNwidth,2+width);
    XtSetValues(widgets[4],args,ONE);
}
XtPopup(shell,XtGrabExclusive);
}

void  FramePointYN(w,closure,call_data)

Widget      w;
caddr_t     closure, call_data;

```

- 123 -

```

{
    Frame frame=(Frame)closure;
    Arg args[1];
    Pixmap pixmap;
    Display *dpy=XtDisplay(global->toplevel);
    Icon point_y=FindIcon("point_y"),
        point_n=FindIcon("point_n");

    Dprintf("FramePointYN\n");
    frame->point_switch=!frame->point_switch;
    XtSetSensitive(frame->image_widget,frame->point_switch);
    XtSetArg(args[0],XtNbitmap,(frame->point_switch?point_y:point_n)->pixmap);
    XtSetValues(w,args,ONE);
    XtSetArg(args[0],XtNbitmap,&pixmap);
    XtGetValues(frame->image_widget,args,ONE);
    UpdatePoint(dpy,frame,pixmap);
    XtSetArg(args[0],XtNbitmap,pixmap);
    XtSetValues(frame->image_widget,args,ONE);
    if (frame->msg!=NULL) UpdateInfo(frame);
}

```

```
void NewPoint(w,closure,call_data)
```

```
Widget w;
```

```
caddr_t closure, call_data;
```

```

{
    Frame frame=(Frame)closure;
    Video vid=frame->video;
    void UpdateFrames();
    int *posn=(int *)call_data,
    channel=frame->channel==3?0:frame->channel;

```

- 124 -

```

    posn[0]=posn[0]>>frame->zoom; posn[1]=posn[1]>>frame->zoom;
    if (vid->trans.type==TRANS_Wave) {
        int    octs=vid->trans.wavelet.space[vid->type==YUV &&
channel!=0?1:0], oct;

    CvIndex(posn[0],posn[1],Size(vid,channel,0),Size(vid,channel,1),octs,&posn[0],&posn[1]
,&oct);
    }
    if (vid->type==YUV && channel!=0) {
        posn[0]=posn[0]<<vid->UVsample[0];
        posn[1]=posn[1]<<vid->UVsample[1];
    }
    Dprintf("NewPoint %d %d previous %d
%d\n",posn[0],posn[1],frame->point->location[0],frame->point->location[1]);
    if (posn[0]!=frame->point->location[0] ||
posn[1]!=frame->point->location[1]) {
        UpdateFrames(global->frames,frame->point,False);
        frame->point->location[0]=posn[0];
        frame->point->location[1]=posn[1];
        UpdateFrames(global->frames,frame->point,True);
    } else Dprintf("No movement\n");
}

void  UpdateFrames(frame,point,update)

Frame frame;
Point point;
Boolean    update;

{
    Arg    args[1];

```

- 125 -

```

    if (frame!=NULL) {
        if (point==frame->point && frame->point_switch==True) {
            Pixmap      pixmap;
            Display      *dpy=XtDisplay(global->toplevel);

            XtSetArg(args[0],XtNbitmap,&pixmap);
            XtGetValues(frame->image_widget,args,ONE);
            UpdatePoint(dpy,frame,pixmap);
            if (update==True) {
                XtSetArg(args[0],XtNbitmap,pixmap);
                XtSetValues(frame->image_widget,args,ONE);
                if (frame->msg!=NULL) UpdateInfo(frame);
            }
        }
        UpdateFrames(frame->next,point,update);
    }
}

```

```

void CloseInfo(w,closure,call_data)

```

```

Widget      w;
caddr_t      closure, call_data;

{
    Frame frame=(Frame)closure;

    frame->msg=NULL;
}

```

```

#define      INFO_ICONS      2

```

```

void FrameInfo(w,closure,call_data)

```


- 126 -

```

Widget      w;
caddr_t     closure, call_data;

{
    Frame frame=(Frame)closure;
    Message   msg=NewMessage(NULL,1000);
    XtCallbackRec   callbacks[]={
        {SetSensitive,(caddr_t)w},
        {CloseInfo,(caddr_t)frame},
        {CloseMessage,(caddr_t)msg},
        {NULL,NULL},
    };
    Dprintf("FrameInfo\n");
    frame->msg=msg;
    UpdateInfo(frame);
    TextSize(msg);
    MessageWindow(w,msg,frame->video->name,True,callbacks);
    XtSetSensitive(w,False);
}

```

```

void  FrameMerge(w,closure,call_data)

```

```

Widget      w;
caddr_t     closure, call_data;

{
    Frame frame=(Frame)closure;
    void  MergePoints();
    Arg   args[1];

    Dprintf("FrameMerge\n");
    MergePoints(global->frames.frame);
}

```

- 127 -

}

```
void MergePoints(frame_search, frame_found)
```

```
Frame frame_search, frame_found;
```

{

```
Arg args[1];
```

```
if (frame_search != NULL) {
```

```
    if (NULL == XawToggleGetCurrent(frame_search->point_merge_widget)
```

```
    || frame_search == frame_found)
```

```
        MergePoints(frame_search->next, frame_found);
```

```
    else {
```

```
        Pixmap pixmap;
```

```
        Display *dpy = XtDisplay(global->toplevel);
```

```
        XtSetArg(args[0], XtNbitmap, &pixmap);
```

```
        XtGetValues(frame_found->image_widget, args, ONE);
```

```
        if (frame_found->point_switch == True)
```

```
UpdatePoint(dpy, frame_found, pixmap);
```

```
        frame_search->point->usage++;
```

```
        frame_found->point->usage--;
```

```
        if (frame_found->point->usage == 0)
```

```
CleanUpPoints(&global->points);
```

```
        frame_found->point = frame_search->point;
```

```
        if (frame_found->point_switch == True) {
```

```
            UpdatePoint(dpy, frame_found, pixmap);
```

```
            XtSetArg(args[0], XtNbitmap, pixmap);
```

```
            XtSetValues(frame_found->image_widget, args, ONE);
```

```
        }
```

```
        if (frame_found->msg != NULL) UpdateInfo(frame_found);
```

- 128 -

```

        XawToggleUnsetCurrent(frame_search->point_merge_widget);
        XawToggleUnsetCurrent(frame_found->point_merge_widget);
    }
}

```

```

#define      POST_DIR      "postscript"

```

```

void  PostScript(w,closure,call_data)

```

```

Widget      w;

```

```

caddr_t      closure, call_data;

```

```

{
    Frame frame=(Frame)closure;
    Video video=frame->video;
    FILE *fp, *fopen();
    char  file_name[STRLEN], hostname[STRLEN];
    int   x, y, width=Size(video,frame->channel,0),
height=Size(video,frame->channel,1);
    struct passwd *pswd;
    long  clock;

    Dprintf("PostScript\n");
    sprintf(file_name, "%s%s/%s.ps\0",global->home,POST_DIR,video->name);
    fp=fopen(file_name,"w");
    fprintf(fp,"%!PS-Adobe-1.0\n");
    pswd = getpwuid (getuid ());
    (void) XmuGetHostname (hostname, sizeof hostname);
    fprintf(fp,"%%%s%%Creator: %s:%s (%s)\n", hostname,pswd->pw_name,
pswd->pw_gecos);
    fprintf(fp,"%%%s%%Title: %s\n",video->name);

```

- 129 -

```

fprintf(fp,"%%%BoundingBox: 0 0 %d %d\n",width,height);
fprintf(fp,"%%%CreationDate: %s",(time (&clock), ctime (&clock)));
fprintf(fp,"%%%EndComments\n");
fprintf(fp,"%d %d scale\n",width,height);
fprintf(fp,"%d %d 8 image_print\n",width,height);
GetFrame(video,frame->frame);
for(y=0;y < height;y++) {
    for(x=0;x < width;x++) {
        int    X, Y, oct, data;

        if (video->trans.type == TRANS_Wave) {

CvtIndex(x,y,width,height,video->trans.wavelet.space[0],&X,&Y,&oct);

data=128+Round(video->data[frame->channel%3][frame->frame][Y*video->size[0]+
X]*(oct == video->trans.wavelet.space[0]?1:4),video->precision);
        } else
data=128+Round(video->data[frame->channel%3][frame->frame][y*video->size[0]+
x],video->precision);
        fprintf(fp,"%02x",data < 0?0:data > 255?255:data);
    }
    fprintf(fp,"\n");
}
FreeFrame(video,frame->frame);
fclose(fp);
}

void Spectrum(w,closure,call_data)

Widget      w;
caddr_t     closure, call_data;

```

- 130 -

```
{
    Frame frame=(Frame)closure;
    Display      *dpy=XtDisplay(global->toplevel);
    XColor       xcolor[2], falsecolor;
    int          i;
    Colormap
cmap=ChannelCmap(frame->channel,frame->video->type,frame->video->gamma);

    Dprintf("Spectrum\n");
    falsecolor.flags=DoRed|DoGreen|DoBlue;
    XSynchronize(dpy,True);
    for(i=0;i<2+global->levels;i++) {
        if (i>1) XStoreColor(dpy,cmap,&xcolor[i&1]); /* Restore old color */
        if (i<global->levels) {
            xcolor[i&1].pixel=i;
            XQueryColor(dpy,cmap,&xcolor[i&1]);
            falsecolor.pixel=i;
            falsecolor.red=xcolor[i&1].red+32512;
            falsecolor.green=xcolor[i&1].green+32512;
            falsecolor.blue=xcolor[i&1].blue+32512;
            XStoreColor(dpy,cmap,&>falsecolor);
        }
    }
    XSynchronize(dpy,False);
}
```

- 131 -

source/icon3.c

```
/*
    Create Icons/Menus and set Callbacks
*/

#include    "../include/xwave.h"

/*    Function Name:    FindIcon
 *    Description:    Finds IconRec entry from name in global icon array
 *    Arguments:    icon_name - name of icon bitmap
 *    Returns:    pointer to IconRec with the same name as icon_name
 */

Icon FindIcon(icon_name)

String icon_name;

{
    int    i;
    Icon    icon=NULL;

    for (i=0;i<global->no_icons;i++)
        if (!strcmp(global->icons[i].name,icon_name)) icon=&global->icons[i];
    return(icon);
}

void FillForm(parent,number,items,widgets,callbacks)

int    number;
```

- 132 -

```

FormItem    items[];
Widget      parent, widgets[];
XtCallbackRec  callbacks[];

{

    Arg    args[10];
    int    i, call_i=0;

    for(i=0;i<number;i++) {
        int    argc=0, *view=(int *)items[i].hook;
        char    text[STRLEN];
        float    top;
        NumInput    num=(NumInput)items[i].hook;
        FloatInput    flt=(FloatInput)items[i].hook;
        Message    msg=(Message)items[i].hook;
        WidgetClass

class[15]={labelWidgetClass,commandWidgetClass,commandWidgetClass,asciiTextWidgetClass,

menuButtonWidgetClass,menuButtonWidgetClass,viewportWidgetClass,toggleWidgetClass

commandWidgetClass,commandWidgetClass,commandWidgetClass,labelWidgetClass,
        scrollbarWidgetClass,labelWidgetClass,formWidgetClass};

        Boolean

call[15]={False,True,True,False,False,False,False,True,True,True,True,False,False,False,False};

        if (items[i].fromHoriz!=0) {
            XtSetArg(args[argc],XtNfromHoriz,widgets[items[i].fromHoriz-1]);

            argc++;
        }
    }
}

```

- 133 -

```

        if (items[i].fromVert!=0) {
            XtSetArg(args[argc],XtNfromVert,widgets[items[i].fromVert-1]);
            argc++;
        }
        switch(items[i].type) { /* Initialise contents */
        case FW_yn:
            items[i].contents=*(Boolean *)items[i].hook?"confirm":"cancel";
            break;
        case FW_up:
            items[i].contents="up";
            break;
        case FW_down:
            items[i].contents="down";
            break;
        case FW_integer:
            sprintf(text,num->format,*num->value);
            items[i].contents=text;
            break;
        case FW_float:
            sprintf(text,flt->format,*flt->value);
            items[i].contents=text;
            break;
        }
        switch(items[i].type) { /* Set contents */
        case FW_label: case FW_command: case FW_button: case FW_integer:
        case FW_float:
            XtSetArg(args[argc],XtNlabel,items[i].contents); argc++;
            break;
        case FW_down: case FW_up: case FW_yn: case FW_toggle: case
        FW_icon: case FW_icon_button: {
            Icon icon=FindIcon(items[i].contents);

```


- 134 -

```

        if (icon == NULL) {
            XtSetArg(args[argc], XtNlabel, items[i].contents); argc++;
        } else {
            XtSetArg(args[argc], XtNbitmap, icon->pixmap); argc++;
            XtSetArg(args[argc], XtNheight, icon->height+2); argc++;
            XtSetArg(args[argc], XtNwidth, icon->width+2); argc++;
        }
        } break;
    }
    switch(items[i].type) { /* Individual set-ups */
    case FW_text:
        XtSetArg(args[argc], XtNstring, msg->info.ptr); argc++;
        XtSetArg(args[argc], XtNeditType, msg->edit); argc++;
        XtSetArg(args[argc], XtNuseStringInPlace, True); argc++;
        XtSetArg(args[argc], XtNlength, msg->size); argc++;
        break;
    case FW_button: case FW_icon_button:
        XtSetArg(args[argc], XtNmenuName, (String)items[i].hook);
        argc++;
        break;
    case FW_toggle:
        if ((int)items[i].hook == 0) {
            XtSetArg(args[argc], XtNradioData, 1); argc++;
        } else {
            caddr_t radioData;
            Arg    radioargs[1];
            Widget  radioGroup = widgets[(int)items[i].hook-1];

            XtSetArg(radioargs[0], XtNradioData, &radioData);
            XtGetValues(radioGroup, radioargs, ONE);

            XtSetArg(args[argc], XtNradioData, (caddr_t)((int)radioData+1)); argc++;

```

- 135 -

```

        XtSetArg(args[argc],XtNradioGroup,radioGroup); argc++;
    }
    break;
case FW_scroll:
    top=(float)(*flt->value-flt->min)/(flt->max-flt->min);
    XtSetArg(args[argc],XtNtopOfThumb,&top); argc++;
    XtSetArg(args[argc],XtNjumpProc,&callbacks[call_i]); argc++;
    while(callbacks[call_i].callback!=NULL) call_i++;
    call_i++;
    break;
case FW_view:
    if (view!=NULL) {
        XtSetArg(args[argc],XtNwidth,view[0]); argc++;
        XtSetArg(args[argc],XtNheight,view[1]); argc++;
    }
    break;
}

widgets[i]=XtCreateManagedWidget(items[i].name,class[(int)items[i].type],parent,args,argc);

switch(items[i].type) { /* Post processing */
case FW_toggle:
    if (items[i].hook==NULL) { /* Avoids Xaw bug */
        XtSetArg(args[0],XtNradioGroup,widgets[i]);
        XtSetValues(widgets[i],args,ONE);
    }
    break;
case FW_text: {
    XFontStruct *font;
    Arg text_args[1];

    msg->widget=widgets[i];

```

- 136 -

```

XawTextDisplayCaret(msg->widget,msg->edit!=XawtextRead);
XtSetArg(text_args[0],XtNfont,&font);
XtGetValues(widgets[i],text_args,ONE);
argc=0;
if (msg->edit==XawtextRead && msg->info.ptr[0]!='\0')
XtSetArg(args[argc],XtNwidth,4+TextWidth(0,msg->info.ptr,font));
else
XtSetArg(args[argc],XtNwidth,4+msg->cols*(font->max_bounds.width+font->min_bo
unds.width)/2);

argc++;

XtSetArg(args[argc],XtNheight,1+msg->rows*(font->max_bounds.ascent+font->max_
bounds.descent)); argc++;
XtSetValues(widgets[i],args,argc);
} break;
case FW_button:

XtOverrideTranslations(widgets[i],XtParseTranslationTable("<BtnDown>: reset()
NameButton() PopupMenu()"));
break;
case FW_down:
if (*num->value==num->min) XtSetSensitive(widgets[i],False);
num->widgets[0]=widgets[i];
break;
case FW_up:
if (*num->value==num->max) XtSetSensitive(widgets[i],False);
num->widgets[1]=widgets[i];
break;
case FW_integer:
num->widgets[2]=widgets[i];
break;
case FW_scroll:

```

- 137 -

```

        flt->widgets[1]=widgets[i];
        XawScrollbarSetThumb(widgets[i],top,0.05);
        break;
    case FW_float:
        flt->widgets[0]=widgets[i];
        break;
    }
    if (call[(int)items[i].type]) { /* Add Callbacks */
        if (callbacks[call_i].callback!=NULL)
            XtAddCallbacks(widgets[i],XtNcallback,&callbacks[call_i]);
        while(callbacks[call_i].callback!=NULL) call_i++;
        call_i++;
    }
}
}

```

```

Widget      ShellWidget(name,parent,type,cmap,callbacks)

```

```

String name;

```

```

Widget      parent;

```

```

ShellWidgetType type;

```

```

Colormap     cmap;

```

```

XtCallbackRec callbacks[];

```

```

{

```

```

    Widget      shell;

```

```

    Arg  args[3];

```

```

    Position    x, y;

```

```

    Dimension    height=-2;

```

```

    int  argc=0;

```

```

    WidgetClass

```

```

class[] = {transientShellWidgetClass,transientShellWidgetClass,topLevelShellWidgetClass,p

```

- 138 -

```
ullRightMenuWidgetClass};
```

```

    if (type == SW_below || type == SW_over) {
        XtTranslateCoords(parent, 0, 0, &x, &y);
        if (type == SW_below) {
            XtSetArg(args[0], XtNheight, &height);
            XtGetValues(parent, args, ONE);
        }
        XtSetArg(args[argc], XtNx, x); argc++;
        XtSetArg(args[argc], XtNy, y + height + 2); argc++;
    }
    if (cmap != NULL) {
        XtSetArg(args[argc], XtNcolormap, cmap); argc++;
    }
    shell = XtCreatePopupShell(name, class[type], parent, args, argc);
    if (callbacks != NULL) XtAddCallbacks(shell, XtNdestroyCallback, callbacks);
    return(shell);
}

```

```
Widget      FormatWidget(name, parent)
```

```
String name;
```

```
Widget      parent;
```

```

{
    return(XtCreateManagedWidget(name, formWidgetClass, parent, NULL, ZERO));
}

```

```
void FillMenu(parent, number, items, widgets, callbacks)
```

```
int number;
```

```
MenuItem items[];
```

- 139 -

```

Widget      parent, widgets[];
XtCallbackRec  callbacks[];

{
    Arg      args[4];
    int      i, call_i=0;
    Icon      icon=FindIcon("right");

    for(i=0;i<number;i++) {
        int      argc=0;

        XtSetArg(args[argc],XtNlabel,items[i].label); argc++;
        if (items[i].widgetClass == smeBSBprObjectClass) {
            XtSetArg(args[argc],XtNmenuName,items[i].hook); argc++;
            XtSetArg(args[argc],XtNrightMargin,4 + icon->width); argc++;
            XtSetArg(args[argc],XtNrightBitmap,icon->pixmap); argc++;
        }

        widgets[i] = XtCreateManagedWidget(items[i].name,items[i].widgetClass,parent,args,argc)
        ;

        if (items[i].widgetClass == smeBSBObjectClass) { /* Add Callbacks */
            XtAddCallbacks(widgets[i],XtNcallback,&callbacks[call_i]);
            while(callbacks[call_i].callback!=NULL) call_i++;
            call_i++;
        }
    }
}

void SimpleMenu(w,closure,call_data)

Widget      w;
caddr_t      closure, call_data;

```

- 140 -

```

{
    int      *hook=(int *)closure, no_child, child, argc=0;
    Widget      menu=XtParent(w), button;
    WidgetList  children;
    char      *label;
    Arg      args[3];

    XtSetArg(args[argc],XtNlabel,&label); argc++;
    XtGetValues(w,args,argc); argc=0;
    XtSetArg(args[argc],XtNchildren,&children); argc++;
    XtSetArg(args[argc],XtNnumChildren,&no_child); argc++;
    XtSetArg(args[argc],XtNbutton,&button); argc++;
    XtGetValues(menu,args,argc); argc=0;
    for(child=0;children[child]!=w && child<no_child;) child++;
    if (w!=children[child]) Eprintf("SimpleMenu: menu error\n");
    *hook=child;
    XtSetArg(args[argc],XtNlabel,label); argc++;
    XtSetValues(button,args,argc);
}

```

```

void  NumIncDec(w,closure,call_data)

```

```

Widget      w;
caddr_t      closure, call_data;

```

```

{
    NumInput  data=(NumInput)closure;
    Arg      args[1];
    char      text[STRLEN];

    *data->value += (w==data->widgets[0])?-1:1;
    sprintf(text,data->format,*data->value);

```

- 141 -

```

    if (data->min == *data->value) XtSetSensitive(data->widgets[0],False);
    else XtSetSensitive(data->widgets[0],True);
    if (data->max == *data->value) XtSetSensitive(data->widgets[1],False);
    else XtSetSensitive(data->widgets[1],True);
    XtSetArg(args[0],XtNlabel,text);
    XtSetValues(data->widgets[2],args,ONE);
}

```

```

void FloatIncDec(w,closure,call_data)

```

```

Widget      w;
caddr_t     closure, call_data;

{
    FloatInput data=(FloatInput)closure;
    Arg  args[1];
    char text[STRLEN];
    float percent=*(float *)call_data;

    *data->value=data->min+(double)percent*(data->max-data->min);
    sprintf(text,data->format,*data->value);
    XtSetArg(args[0],XtNlabel,text);
    XtSetValues(data->widgets[0],args,ONE);
}

```

```

/*  Function Name:      ChangeYN
 *
 *  Description:  Toggle YN widget state
 *
 *  Arguments:  w - toggling widget
 *
 *              closure - pointer to boolean state
 *
 *              call_data - not used
 *
 *  Returns:      none.
 */

```


- 142 -

```
void ChangeYN(w,closure,call_data)
```

```
Widget      w;
```

```
caddr_t     closure, call_data;
```

```
{
```

```
    Boolean      *bool=(Boolean *)closure;
```

```
    Icon   icon=FindIcon((*bool != True)? "confirm": "cancel");
```

```
    Arg   args[4];
```

```
    int   argc=0;
```

```
    *bool = ! *bool;
```

```
    XtSetArg(args[argc],XtNbitmap,icon->pixmap); argc ++;
```

```
    XtSetArg(args[argc],XtNheight,icon->height+2); argc ++;
```

```
    XtSetArg(args[argc],XtNwidth,icon->width+2); argc ++;
```

```
    XtSetValues(w,args,argc);
```

```
}
```

```
int TextWidth(max,text,font)
```

```
int max;
```

```
String text;
```

```
XFontStruct *font;
```

```
{
```

```
    int   i=0, j;
```

```
    while(text[i]!='\0') {
```

```
        int   width;
```

```
        for(j=0;text[i+j]!='\0' && text[i+j]!='\n';) j ++;
```

```
        width=XTextWidth(font,&text[i],j);
```

- 143 -

max = max > width?max:width;

/******

Copyright 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts,
and the Massachusetts Institute of Technology, Cambridge, Massachusetts.

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the names of Digital or MIT not be
used in advertising or publicity pertaining to distribution of the
software without specific, written prior permission.

DIGITAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING

ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL

DIGITAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL
DAMAGES OR

ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
PROFITS,

WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
TORTIOUS ACTION,

ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
THIS
SOFTWARE.

*****/

- 144 -

```

/*
 * Image.c - Image widget
 *
 */

#define XtStrlen(s)      ((s) ? strlen(s) : 0)

#include <stdio.h>
#include <ctype.h>
#include <X11/IntrinsicP.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/XawInit.h>
#include "../include/ImageP.h"

#define streq(a,b) (strcmp( (a), (b) ) == 0)

/*****
 *
 * Full class record constant
 *
 *****/

/* Private Data */

static char defaultTranslations[] =
    "<Btn1Down>: notify()\n\
    <Btn1Motion>: notify()\n\
    <Btn1Up>: notify()";

#define offset(field) XtOffset(ImageWidget, field)

static XtResource resources[] = {
    {XtNbitmap, XtCPixmap, XtRBitmap, sizeof(Pixmap),

```

- 145 -

```

        offset(image.pixmap), XtRImmediate, (caddr_t)None},
        {XtNcallback, XtCCallback, XtRCallback, sizeof(XtPointer),
        offset(image.callbacks), XtRCallback, (XtPointer)NULL},
    };

```

```

static void Initialize();
static void Resize();
static void Redisplay();
static Boolean SetValues();
static void ClassInitialize();
static void Destroy();
static XtGeometryResult QueryGeometry();

```

```

static void Notify(), GetBitmapInfo();

```

```

static XtActionsRec      actionsList[]={
    {"notify",    Notify},
};

```

```

ImageClassRec imageClassRec = {
    {
/* core_class fields */
#define superclass      (&simpleClassRec)
        /* superclass      */ (WidgetClass) superclass,
        /* class_name      */ "Image",
        /* widget_size     */ sizeof(ImageRec),
        /* class_initialize */ ClassInitialize,
        /* class_part_initialize */ NULL,
        /* class_init      */ FALSE,
        /* initialize      */ Initialize,
        /* initialize_hook  */ NULL,
        /* realize         */ XtInheritRealize,

```

- 146 -

```

/* actions          */  actionsList,
/* num_actions      */  XtNumber(actionsList),
/* resources        */  resources,
/* num_resources    */  XtNumber(resources),
/* xrm_class        */  NULLQUARK,
/* compress_motion  */  TRUE,
/* compress_exposure */  TRUE,
/* compress_enterleave */  TRUE,
/* visible_interest */  FALSE,
/* destroy          */  Destroy,
/* resize           */  Resize,
/* expose          */  Redisplay,
/* set_values       */  SetValues,
/* set_values_hook  */  NULL,
/* set_values_almost */  XtInheritSetValuesAlmost,
/* get_values_hook  */  NULL,
/* accept_focus     */  NULL,
/* version          */  XtVersion,
/* callback_private */  NULL,
/* tm_table         */  defaultTranslations,
/* query_geometry   */  QueryGeometry,
/* display_accelerator */  XtInheritDisplayAccelerator,
/* extension        */  NULL
},
/* Simple class fields initialization */
{
/* change_sensitive */  XtInheritChangeSensitive
}
};

WidgetClass imageWidgetClass = (WidgetClass)&imageClassRec;
/*****

```

- 147 -

```

* Private Procedures
*
*****/

static void ClassInitialize()
{
    extern void XmuCvtStringToBitmap();
    static XtConvertArgRec screenConvertArg[] = {
        {XtWidgetBaseOffset, (caddr_t) XtOffset(Widget, core.screen),
         sizeof(Screen *)}
    };
    XawInitializeWidgetSet();
    XtAddConverter("String", "Bitmap", XmuCvtStringToBitmap,
                  screenConvertArg, XtNumber(screenConvertArg));
} /* ClassInitialize */

/* ARGSUSED */
static void Initialize(request,new)

Widget request, new;

{
    ImageWidget iw = (ImageWidget) new;

    Dprintf("ImageInitialize\n");
    if (iw->image.pixmap == NULL)
        XtErrorMsg("NoBitmap", "asciiSourceCreate", "XawError",
                  "Image widget has no bitmap.", NULL, 0);
    GetBitmapInfo(new);
    if (iw->image.map_width <= 0 || iw->image.map_height <= 0)
        XtErrorMsg("NoDimension", "asciiSourceCreate", "XawError",
                  "Image widget illegal map dimension.", NULL, 0);
}

```

- 148 -

```
if (iw->core.width == 0) iw->core.width=iw->image.map_width;
    if (iw->core.height == 0) iw->core.height=iw->image.map_height;

(*XtClass(new)->core_class.resize) ((Widget)iw);

} /* Initialize */

/*
 * Repaint the widget window
 */

/* ARGSUSED */
static void Redisplay(w, event, region)
    Widget w;
    XEvent *event;
    Region region;
{
    ImageWidget iw = (ImageWidget) w;

    Dprintf("ImageRedisplay\n");
    if (region != NULL &&
        XRectInRegion(region, 0, 0,
            iw->image.map_width, iw->image.map_height)
        == RectangleOut)
        return;

    XCopyArea(
        XtDisplay(w), iw->image.pixmap, XtWindow(w),
        DefaultGC(XtDisplay(w),XDefaultScreen(XtDisplay(w))),
        0, 0, iw->image.map_width, iw->image.map_height, 0, 0);
}
```

- 149 -

```
static void Resize(w)
    Widget w;
{
    ImageWidget iw = (ImageWidget)w;
    Dprintf("ImageResize\n");
}

/*
 * Set specified arguments into widget
 */

static Boolean SetValues(current, request, new, args, num_args)
    Widget current, request, new;
    ArgList args;
    Cardinal *num_args;
{
    ImageWidget curiw = (ImageWidget) current;
    ImageWidget reqiw = (ImageWidget) request;
    ImageWidget newiw = (ImageWidget) new;
    Boolean redisplay = False;

    /* recalculate the window size if something has changed. */

    if (curiw->image.pixmap != newiw->image.pixmap)
XFreePixmap(XtDisplay(curiw),curiw->image.pixmap);
    GetBitmapInfo(newiw);
    newiw->core.width = newiw->image.map_width;
    newiw->core.height = newiw->image.map_height;
    redisplay = True;

    return redisplay || XtIsSensitive(current) != XtIsSensitive(new);
}
```


- 150 -

```
static void Destroy(w)
    Widget w;
{
    ImageWidget iw = (ImageWidget)w;

    Dprintf("ImageDestroy\n");
}

static XtGeometryResult QueryGeometry(w, intended, preferred)
    Widget w;
    XtWidgetGeometry *intended, *preferred;
{
    register ImageWidget iw = (ImageWidget)w;

    preferred->request_mode = CWWidth | CWHeight;
    preferred->width = iw->image.map_width;
    preferred->height = iw->image.map_height;
    if ( ((intended->request_mode & (CWWidth | CWHeight))
         == (CWWidth | CWHeight)) &&
        intended->width == preferred->width &&
        intended->height == preferred->height)
        return XtGeometryYes;
    else if (preferred->width == w->core.width &&
             preferred->height == w->core.height)
        return XtGeometryNo;
    else
        return XtGeometryAlmost;
}

static void GetBitmapInfo(w)
```

- 151 -

```

Widget      w;

{
    ImageWidget iw=(ImageWidget)w;
    unsigned int  depth, bw;
    Window      root;
    int         x, y;
    unsigned int  width, height;
    char        buf[BUFSIZ];

    if (iw->image.pixmap != None) {
        if
(!XGetGeometry(XtDisplayOfObject(w),iw->image.pixmap,&root,&x,&y,&width,&height,&bw,&depth)) {
            sprintf(buf, "ImageWidget: %s %s \"%s\".", "Could not",
            "get Bitmap geometry information for Image ",
            XtName(w));
            XtAppError(XtWidgetToApplicationContext(w), buf);
        }
        iw->image.map_width=(Dimension)width;
        iw->image.map_height=(Dimension)height;
    }
}

/*
 *   Action Procedures
 */

static void  Notify(w,event,params,num_params)

Widget      w;
XEvent      *event;

```

- 152 -

```
String *params;
Cardinal      *num_params;

{
    ImageWidget iw=(ImageWidget)w;
    XButtonEvent *buttonevent=&event->xbutton;
    int    posn[2]={buttonevent->x,buttonevent->y};

    if (iw->image.map_width <= posn[0] || posn[0] < 0 ||
        iw->image.map_height <= posn[1] || posn[1] < 0) Dprintf("No
ImageNotify\n");
    else {
        Dprintf("ImageNotify\n");
        XtCallCallbackList(w,iw->image.callbacks,posn);
    }
}
```

- 153 -

source/ImpKlicsTestSA.c

```
/*
    Test harness for KlicsFrameSA() in Klics.SA
*/

#include    "xwave.h"
#include    "KlicsSA.h"

void  ImpKlicsTestSA(w,closure,call_data)

Widget      w;
caddr_t     closure, call_data;

{
    int      sizeY=SA_WIDTH*SA_HEIGHT,
              sizeUV=SA_WIDTH*SA_HEIGHT/4;
    short    *dst[3]={
        (short *)MALLOC(sizeof(short)*sizeY),
        (short *)MALLOC(sizeof(short)*sizeUV),
        (short *)MALLOC(sizeof(short)*sizeUV),
    }, *src[3];
    Video    video=(Video)MALLOC(sizeof(VideoRec));
    int      i, z;
    char      file_name[STRLEN];
    Bits      bfp;
    Boolean    stillvid;

    strcpy(video->name,((XawListReturnStruct *)call_data)->string);
```

```
sprintf(file_name, "%s%s/ %s%s\0", global->home, KLICS_SA_DIR, video->name, KLICS_SA_EXT);
```

```
    bfp = bopen(file_name, "r");
    bread(&stillvid, 1, bfp);
    bread(&video->size[2], sizeof(int)*8, bfp);
    video->data[0] = (short **)MALLOC(sizeof(short *)*video->size[2]);
    video->data[1] = (short **)MALLOC(sizeof(short *)*video->size[2]);
    video->data[2] = (short **)MALLOC(sizeof(short *)*video->size[2]);
    video->disk = False;
    video->type = YUV;
    video->size[0] = SA_WIDTH;
    video->size[1] = SA_HEIGHT;
    video->UVsample[0] = 1;
    video->UVsample[1] = 1;
    video->trans.type = TRANS_None;
    for(z=0; z<video->size[2]; z++) {
        NewFrame(video, z);
        src[0] = video->data[0][z];
        src[1] = video->data[1][z];
        src[2] = video->data[2][z];
        KlicsFrameSA(z == 0 || stillvid?STILL:SEND, src, dst, bfp);
        SaveFrame(video, z);
        FreeFrame(video, z);
    }
    bclose(bfp);
    video->next = global->videos;
    global->videos = video;
    XtFree(dst[0]);
    XtFree(dst[1]);
    XtFree(dst[2]);
```

```
}
```

- 155 -

source/ImportKlics.c

```
/*
 *   Importing raw Klics binary files
 */

#include    "xwave.h"
#include    "Klics.h"

extern Bits  bopen();
extern void  bclose(), bread(), bwrite(), bflush();

extern void  SkipFrame();
extern int   HuffRead();
extern Boolean  BlockZero();
extern void  ZeroCoeffs();
extern int   ReadInt();
extern int   Decide();
extern double  DecideDouble();

Boolean  BoolToken(bfp)

Bits  bfp;

{
    Boolean  token;

    bread(&token,1,bfp);
    return(token);
}
```

- 156 -

```
void HuffBlock(block,bfp)
```

```
Block block;
```

```
Bits bfp;
```

```
{
    int X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
        block[X][Y]=HuffRead(bfp);
}
```

```
void PrevBlock(old,addr,x,y,z,oct,sub,channel,ctrl)
```

```
Block old, addr;
```

```
int x, y, z, oct, sub, channel;
```

```
CompCtrl ctrl;
```

```
{
    int X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++) {

        addr[X][Y]=Access((x<<1)+X,(y<<1)+Y,oct,sub,Size(ctrl->dst,channel,0));
        old[X][Y]=ctrl->dst->data[channel][z][addr[X][Y]];
    }
}
```

```
void DeltaBlock(new,old,delta,step)
```

```
Block new, old, delta;
```

```
int step;
```

- 157 -

```

{
    int    X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)

new[X][Y]=old[X][Y]+delta[X][Y]*step+(delta[X][Y]!=0?negif(delta[X][Y]<0,(step-1)
>>1):0);
}

```

```

void  UpdateBlock(new,addr,z,channel,ctrl)

```

```

int    z, channel;
Block new, addr;
CompCtrl  ctrl;

```

```

{
    int    X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
        ctrl->dst->data[channel][z][addr[X][Y]]=(short)new[X][Y];
}

```

```

void  ReadKlicsHeader(ctrl)

```

```

CompCtrl  ctrl;

```

```

{
    KlicsHeaderRec  head;
    int    i;
    Video dst=ctrl->dst;

    fread(&head,sizeof(KlicsHeaderRec),1,ctrl->bfp->fp);

```


- 158 -

```

ctrl->stillvid= head.stillvid;
ctrl->auto_q= head.auto_q;
ctrl->buf_switch= head.buf_switch;
ctrl->quant_const= head.quant_const;
ctrl->thresh_const= head.thresh_const;
ctrl->cmp_const= head.cmp_const;
ctrl->fps= head.fps;
for(i=0;i<5;i++) ctrl->base_factors[i]= head.base_factors[i];
ctrl->diag_factor= head.diag_factor;
ctrl->chrome_factor= head.chrome_factor;
ctrl->decide= head.decide;
strcpy(dst->name,ctrl->bin_name);
dst->type= head.type;
dst->disk= head.disk;
dst->gamma= head.gamma;
dst->rate= head.rate;
dst->start= head.start;
for(i=0;i<3;i++) dst->size[i]= head.size[i];
for(i=0;i<2;i++) dst->UVsample[i]= head.UVsample[i];
dst->trans= head.trans;
dst->precision= head.precision;
for(i=0;i<(dst->type==MONO?1:3);i++)
    dst->data[i]=(short **)MALLOC(dst->size[2]*sizeof(short *));
}

```

```
void WriteKlicsHeader(ctrl)
```

```
CompCtrl ctrl;
```

```

{
    KlicsHeaderRec head;
    int i;

```

- 159 -

```

    head.stillvid=ctrl->stillvid;
    head.auto_q=ctrl->auto_q;
    head.buf_switch=ctrl->buf_switch;
    head.quant_const=ctrl->quant_const;
    head.thresh_const=ctrl->thresh_const;
    head.cmp_const=ctrl->cmp_const;
    head.fps=ctrl->fps;
    for(i=0;i<5;i++) head.base_factors[i]=ctrl->base_factors[i];
    head.diag_factor=ctrl->diag_factor;
    head.chrome_factor=ctrl->chrome_factor;
    head.decide=ctrl->decide;
    head.type=ctrl->dst->type;
    head.disk=ctrl->dst->disk;
    head.gamma=ctrl->dst->gamma;
    head.rate=ctrl->dst->rate;
    head.start=ctrl->dst->start;
    for(i=0;i<3;i++) head.size[i]=ctrl->dst->size[i];
    for(i=0;i<2;i++) head.UVsample[i]=ctrl->dst->UVsample[i];
    head.trans=ctrl->dst->trans;
    head.precision=ctrl->dst->precision;
    fwrite(&head,sizeof(KlicsHeaderRec),1,ctrl->bfp->fp);
}

```

```
void KlicsTree(mode,x,y,z,oct,sub,channel,ctrl)
```

```
int mode, x, y, z, oct, sub, channel;
```

```
CompCtrl ctrl;
```

```
{
```

```
Block addr, old, new, delta, zero_block={{0,0},{0,0}};
```

```
double norms[3]={ctrl->quant_const,ctrl->thresh_const,ctrl->cmp_const};
```

```
int step;
```

- 160 -

```

PrevBlock(old,addr,x,y,z,oct,sub,channel,ctrl);
if (mode!=VOID) {
    CalcNormals(ctrl,oct,sub,channel,norms);
    step=norms[0] < 1.0?1:(int)norms[0];
    if (mode==STILL || BlockZero(old)) {
        if (BoolToken(ctrl->bfp)) { /* NON_ZERO_STILL */
            Dprintf("NON_ZERO_STILL\n");
            HuffBlock(delta,ctrl->bfp);
            DeltaBlock(new,old,delta,step);
            UpdateBlock(new,addr,z,channel,ctrl);
        } else {
            Dprintf("ZERO_STILL\n");
            mode=STOP; /* ZERO_STILL */
        }
    } else {
        if (!BoolToken(ctrl->bfp)) { /* BLOCK_SAME */
            Dprintf("BLOCK_SAME\n");
            mode=STOP;
        } else {
            if (!BoolToken(ctrl->bfp)) { /* ZERO_VID */
                Dprintf("ZERO_VID\n");
                ZeroCoeffs(ctrl->dst->data[channel][z],addr);
                mode=VOID;
            } else { /*
BLOCK_CHANGE */
                Dprintf("BLOCK_CHANGE\n");
                HuffBlock(delta,ctrl->bfp);
                DeltaBlock(new,old,delta,step);
                UpdateBlock(new,addr,z,channel,ctrl);
            }
        }
    }
}
}

```

- 161 -

```

    } else {
        if (BlockZero(old)) mode = STOP;
        else {
            ZeroCoeffs(ctrl->dst->data[channel][z],addr);
            mode = VOID;
        }
    }
    if (oct > 0 && mode != STOP) {
        Boolean    decend = mode == VOID ? True : BoolToken(ctrl->bfp);
        int    X, Y;

        Dprintf("x = %d, y = %d, oct = %d sub = %d mode\n",x,y,oct,sub,mode);
        if (decend) {
            if (mode != VOID) Dprintf("OCT_NON_ZERO\n");
            for(Y=0;Y<2;Y++) for(X=0;X<2;X++)
                KlicsTree(mode,x*2+X,y*2+Y,z,oct-1,sub,channel,ctrl);
        } else if (mode != VOID) Dprintf("OCT_ZERO\n");
    }
}

void KlicsLPF(mode,z,ctrl)

CompCtrl    ctrl;
int    mode, z;

{
    Block addr, old, new, delta;
    int    channel, channels=ctrl->dst->type == MONO?1:3, x, y,
        octs_lum=ctrl->dst->trans.wavelet.space[0],

    size[2] = {Size(ctrl->dst,0,0) >> octs_lum + 1, Size(ctrl->dst,0,1) >> octs_lum + 1};

```

- 162 -

```

for(y=0;y < size[1];y++) for(x=0;x < size[0];x++) {
    Boolean    lpf_loc = True;

    if (mode != STILL) {
        lpf_loc = BoolToken(ctrl->bfp); /*
LPF_LOC_ZERO/LPF_LOC_NON_ZERO */

Dprintf("%s\n",lpf_loc?"LPF_LOC_NON_ZERO":"LPF_LOC_ZERO");
    }
    if (lpf_loc) for(channel=0;channel < channels;channel++) {
        int
octs = ctrl->dst->trans.wavelet.space[ctrl->dst->type == YUV && channel != 0?1:0],
        X, Y, step, value, bits = 0;

        double
norms[3] = {ctrl->quant_const,ctrl->thresh_const,ctrl->cmp_const};

        PrevBlock(old,addr,x,y,z,octs-1,0,channel,ctrl);
        CalcNormals(ctrl,octs-1,0,channel,norms);
        step = norms[0] < 1.0?1:(int)norms[0];
        if (mode == STILL) {
            for(bits=0,
value = ((1 < 8+ctrl->dst->precision)-1)/step;value != 0;bits++)
                value = value >> 1;
            for(X=0;X < BLOCK;X++) for(Y=0;Y < BLOCK;Y++)
                delta[X][Y] = ReadInt(bits,ctrl->bfp);
            DeltaBlock(new,old,delta,step);
            UpdateBlock(new,addr,z,channel,ctrl);
        } else {
            if (BoolToken(ctrl->bfp)) { /*
LPF_ZERO/LPF_NON_ZERO */

                Dprintf("LPF_NON_ZERO\n");
                HuffBlock(delta,ctrl->bfp);

```

- 163 -

```

        DeltaBlock(new,old,delta,step);
        UpdateBlock(new,addr,z,channel,ctrl);
    } else Dprintf("LPF_ZERO\n");
    }
    }
    }
}

void KlicsFrame(ctrl,z)

CompCtrl    ctrl;
int         z;

{
    Video dst=ctrl->dst;
    int    sub, channel, x, y, mode=ctrl->stillvid || z==0?STILL:SEND,
          octs_lum=dst->trans.wavelet.space[0],

size[2]={Size(dst,0,0)>>1+octs_lum,Size(dst,0,1)>>1+octs_lum};

    NewFrame(dst,z);
    CopyFrame(dst,z-1,z,ctrl->stillvid || z==0);
    if (z!=0 && ctrl->auto_q) {

ctrl->quant_const+=(double)(HISTO/2+ReadInt(HISTO_BITS,ctrl->bfp))*HISTO_DE
LTA*2.0/HISTO-HISTO_DELTA;

        ctrl->quant_const=ctrl->quant_const<0.0?0.0:ctrl->quant_const;
        Dprintf("New quant %f\n",ctrl->quant_const);
    }
    KlicsLPF(mode,z,ctrl);
    for(y=0;y<size[1];y++) for(x=0;x<size[0];x++) {
        if (BoolToken(ctrl->bfp)) {

```

- 164 -

```

Dprintf("LOCAL_NON_ZERO\n");
for(channel=0;channel<(dst->type==MONO?1:3);channel++) {
    int    octs=dst->trans.wavelet.space[dst->type==YUV
&& channel!=0?1:0];

    if (BoolToken(ctrl->bfp)) {
        Dprintf("CHANNEL_NON_ZERO\n");
        for(sub=1;sub<4;sub++)
            KlicsTree(mode,x,y,z,octs-1,sub,channel,ctrl);
    } else Dprintf("CHANNEL_ZERO\n");
}
} else Dprintf("LOCAL_ZERO\n");
}
}

```

```

void ImportKlics(w,closure,call_data)

```

```

Widget      w;

```

```

caddr_t      closure, call_data;

```

```

{

```

```

    char    file_name[STRLEN];

```

```

    CompCtrlRec ctrl;

```

```

    int     i, z;

```

```

    ctrl.dst=(Video)MALLOC(sizeof(VideoRec));

```

```

    strcpy(ctrl.bin_name,((XawListReturnStruct *)call_data)->string);

```

```

sprintf(file_name,"%s%s/%s%s\0",global->home,KLICS_DIR,ctrl.bin_name,KLICS_EX
T);

```

```

    ctrl.bfp=bopen(file_name,"r");

```

```

    ReadKlicsHeader(&ctrl);

```

- 165 -

```
if (ctrl.dst->disk) SaveHeader(ctrl.dst);
for(z=0;z < ctrl.dst->size[2];z++) {
    if (z==0 || !ctrl.buf_switch) KlicsFrame(&ctrl,z);
    else {
        if (BoolToken(ctrl.bfp)) KlicsFrame(&ctrl,z);
        else SkipFrame(ctrl.dst,z);
    }
    if (z > 0) {
        SaveFrame(ctrl.dst,z-1);
        FreeFrame(ctrl.dst,z-1);
    }
}
SaveFrame(ctrl.dst,ctrl.dst->size[2]-1);
FreeFrame(ctrl.dst,ctrl.dst->size[2]-1);
bclose(ctrl.bfp);
ctrl.dst->next=global->videos;
global->videos=ctrl.dst;
}
```


- 166 -

```
source/ImportKlicsSA.c
```

```
*****/
```

```
/*
```

```
 *   Importing raw Klics binary files
```

```
 *
```

```
 *   Stand Alone version
```

```
*/
```

```
#include    "KlicsSA.h"
```

```
extern void  Convolve();
```

```
/* useful X definitions */
```

```
typedef char  Boolean;
```

```
#define True   1
```

```
#define False  0
```

```
#define String char*
```

```
extern int    HuffReadSA();
```

```
extern Boolean  BlockZeroSA();
```

```
extern void    ZeroCoeffsSA();
```

```
extern int     ReadIntSA();
```

```
extern int     DecideSA();
```

```
extern double   DecideDoubleSA();
```

```
Boolean    BoolTokenSA(bfp)
```

```
Bits    bfp;
```

```
{
```

- 167 -

```
Boolean    token;

    bread(&token,1,bfp);
    return(token);
}

void HuffBlockSA(block,bfp)

Block block;
Bits  bfp;

{
    int    X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
        block[X][Y]=HuffReadSA(bfp);
}

void PrevBlockSA(old,addr,x,y,oct,sub,channel,dst)

Block old, addr;
int    x, y, oct, sub, channel;
short *dst[3];

{
    int    X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++) {
        addr[X][Y]=AccessSA((x<<1)+X,(y<<1)+Y,oct,sub,channel);
        old[X][Y]=dst[channel][addr[X][Y]];
    }
}
```

- 168 -

```
void DeltaBlockSA(new,old,delta,step)
```

```
Block new, old, delta;
```

```
int step;
```

```
{
```

```
int X, Y;
```

```
for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
```

```
new[X][Y]=old[X][Y]+delta[X][Y]*step+(delta[X][Y]!=0?negif(delta[X][Y]<0,(step-1)  
>>1):0);
```

```
}
```

```
void UpdateBlockSA(new,addr,channel,dst)
```

```
int channel;
```

```
Block new, addr;
```

```
short *dst[3];
```

```
{
```

```
int X, Y;
```

```
for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
```

```
dst[channel][addr[X][Y]]=(short)new[X][Y];
```

```
}
```

```
void KlicsTreeSA(mode,x,y,oct,sub,channel,dst,bfp,quant_const)
```

```
int mode, x, y, oct, sub, channel;
```

```
short *dst[3];
```

```
Bits bfp;
```

- 169 -

```
double    quant_const;
```

```
{
```

```
Block addr, old, new, delta, zero_block={{0,0},{0,0}};
```

```
double    norms[3]={quant_const,thresh_const,cmp_const};
```

```
int    step;
```

```
PrevBlockSA(old,addr,x,y,oct,sub,channel,dst);
```

```
if (mode!=VOID) {
```

```
    CalcNormalsSA(oct,sub,channel,norms,quant_const);
```

```
    step=norms[0]<1.0?1:(int)norms[0];
```

```
    if (mode==STILL || BlockZero(old)) {
```

```
        if (BoolTokenSA(bfp)) { /* NON_ZERO_STILL */
```

```
            Dprintf("NON_ZERO_STILL\n");
```

```
            HuffBlockSA(delta,bfp);
```

```
            DeltaBlockSA(new,old,delta,step);
```

```
            UpdateBlockSA(new,addr,channel,dst);
```

```
        } else {
```

```
            Dprintf("ZERO_STILL\n");
```

```
            mode=STOP; /* ZERO_STILL */
```

```
        }
```

```
    } else {
```

```
        if (!BoolTokenSA(bfp)) { /* BLOCK_SAME */
```

```
            Dprintf("BLOCK_SAME\n");
```

```
            mode=STOP;
```

```
        } else {
```

```
            if (!BoolTokenSA(bfp)) { /* ZERO_VID */
```

```
                Dprintf("ZERO_VID\n");
```

```
                ZeroCoeffsSA(dst[channel],addr);
```

```
                mode=VOID;
```

```
            } else {
```

```
                /*
```

```
            BLOCK_CHANGE */
```

- 170 -

```

        Dprintf("BLOCK_CHANGE\n");
        HuffBlockSA(delta,bfp);
        DeltaBlockSA(new,old,delta,step);
        UpdateBlockSA(new,addr,channel,dst);
    }
}
}
} else {
    if (BlockZeroSA(old)) mode = STOP;
    else {
        ZeroCoeffsSA(dst[channel],addr);
        mode = VOID;
    }
}
if (oct > 0 && mode != STOP) {
    Boolean    decend = mode == VOID ? True : BoolTokenSA(bfp);
    int    X, Y;

    Dprintf("x = %d, y = %d, oct = %d sub = %d mode .
%d\n",x,y,oct,sub,mode);
    if (decend) {
        if (mode != VOID) Dprintf("OCT_NON_ZERO\n");
        for(Y=0;Y<2;Y++) for(X=0;X<2;X++)

KlicsTreeSA(mode,x*2+X,y*2+Y,oct-1,sub,channel,dst,bfp,quant_const);
    } else if (mode != VOID) Dprintf("OCT_ZERO\n");
}
}

void  KlicsLPF_SA(mode,dst,bfp,quant_const)

int    mode;

```

- 171 -

```

short  *dst[3];
Bits   bfp;
double  quant_const;

{
    Block  addr, old, new, delta;
    int    channel, channels=3, x, y,
           octs_lum=3,

size[2]={SA_WIDTH>>octs_lum+1,SA_HEIGHT>>octs_lum+1};

    for(y=0;y<size[1];y++) for(x=0;x<size[0];x++) {
        Boolean    lpf_loc=True;

        if (mode!=STILL) {
            lpf_loc=BoolTokenSA(bfp); /*
LPF_LOC_ZERO/LPF_LOC_NON_ZERO */

Dprintf("%s\n",lpf_loc?"LPF_LOC_NON_ZERO":"LPF_LOC_ZERO");
        }
        if (lpf_loc) for(channel=0;channel<channels;channel++) {
            int    octs=channel!=0?2:3,
                    X, Y, step, value, bits=0;
            double  norms[3]={quant_const,thresh_const,cmp_const};

            PrevBlockSA(old,addr,x,y,octs-1,0,channel,dst);
            CalcNormalsSA(octs-1,0,channel,norms,quant_const);
            step=norms[0]<1.0?1:(int)norms[0];
            if (mode==STILL) {
                for(bits=0,
value=((1<<8+SA_PRECISION)-1)/step;value!=0;bits++)
                    value=value>>1;

```

- 172 -

```

        for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
            delta[X][Y]=ReadIntSA(bits,bfp);
        DeltaBlockSA(new,old,delta,step);
        UpdateBlockSA(new,addr,channel,dst);
    } else {
        if (BoolTokenSA(bfp)) { /* LPF_ZERO/LPF_NON_ZERO
*/
            Dprintf("LPF_NON_ZERO\n");
            HuffBlockSA(delta,bfp);
            DeltaBlockSA(new,old,delta,step);
            UpdateBlockSA(new,addr,channel,dst);
        } else Dprintf("LPF_ZERO\n");
    }
}

void KlicsFrameSA(mode,src,dst,bfp)

int    mode;
short  *src[3], *dst[3];
Bits   bfp;

{
    int    sub, channel, x, y, i,
           octs_lum=3,

size[2]={SA_WIDTH>>1+octs_lum,SA_HEIGHT>>1+octs_lum};
    double    quant_const;

    bread((char *)&quant_const,sizeof(double)*8,bfp);
    KlicsLPF_SA(mode,dst,bfp,quant_const);

```

- 173 -

```

for(y=0;y<size[1];y++) for(x=0;x<size[0];x++) {
    if (BoolTokenSA(bfp)) {
        Dprintf("LOCAL_NON_ZERO\n");
        for(channel=0;channel<3;channel++) {
            int    octs=channel!=0?2:3;

            if (BoolTokenSA(bfp)) {
                Dprintf("CHANNEL_NON_ZERO\n");
                for(sub=1;sub<4;sub++)

                    KlicsTreeSA(mode,x,y,octs-1,sub,channel,dst,bfp,quant_const);
            } else Dprintf("CHANNEL_ZERO\n");
        }
    } else Dprintf("LOCAL_ZERO\n");
}

for(channel=0;channel<3;channel++) {
    int
    frame_size[2]={SA_WIDTH>>(channel==0?0:1),SA_HEIGHT>>(channel==0?0:1
    )},

    frame_area=frame_size[0]*frame_size[1];

    for(i=0;i<frame_area;i++) src[channel][i]=dst[channel][i];
    Convolve(src[channel],False,frame_size,channel==0?3:2,0);
    for(i=0;i<frame_area;i++)
        src[channel][i]=src[channel][i]>>SA_PRECISION;
}
}

```


- 174 -

source/InitFrame.c

```
/*
    Initialise frame structure for Frame command widget
*/

#include    "../include/xwave.h"

#define     FRAME_ICONS      14
#define     TRANS_MENU      1
#define     COMP_MENU        2


extern void CopyVideo();
extern void Compare();
extern void NA();
extern void FrameDestroy();
extern void Examine();
extern void FramePointYN();
extern void FrameInfo();
extern void FrameMerge();
extern void Movie();
extern void PostScript();
extern void Select();
extern void Spectrum();
extern void NewPoint();
extern void Transform();
extern void Compress();
extern String *VideoCurrentList();
extern void KlicsSA();


void InitFrame    (w,closure,call_data)
```

- 175 -

```

Widget      w;
caddr_t     closure, call_data;

{
    XawListReturnStruct *name=(XawListReturnStruct *)call_data;
    Video video=FindVideo(name->string,global->videos);
    Frame frame=(Frame)MALLOC(sizeof(FrameRec));
    Widget     shell[2], form, widgets[FRAME_ICONS],
trans_widgets[TRANS_MENU], comp_widgets[COMP_MENU];
    Arg  args[7];
    Pixmap  pixmap;
    int  view[2]={15+video->size[0],15+video->size[1]};
    FormItem  items[]={
        {"frm_cancel",      "frame_close",          0,0,FW_icon,NULL},
        {"frm_copy", "copy",          1,0,FW_icon,NULL},
        {"frm_exam",      "examine",          2,0,FW_icon,NULL},
        {"frm_point_yn","point_y",          3,0,FW_icon,NULL},
        {"frm_transform","transform",
4,0,FW_icon_button,"frm_trans_menu"},
        {"frm_info_yn",      "info",
5,0,FW_icon,NULL},
        {"frm_merge",      "merge",          6,0,FW_toggle,NULL},
        {"frm_compress","code",
7,0,FW_icon_button,"frm_comp_menu"},
        {"frm_movie",      "movie",          8,0,FW_icon,NULL},
        {"frm_postscript","postscript",          9,0,FW_icon,NULL},
        {"frm_compare",      "compare",          10,0,FW_icon,NULL},
        {"frm_view", NULL,
0,1,FW_view,(String)view},
        {"frm_label", video->name,          0,12,FW_label,NULL},
        {"frm_colors",      "colors",          13,12,FW_icon,NULL},
    };
};

```

- 176 -

```

Selection    sel=(Selection)MALLOC(sizeof(SelectItem));
MenuItem     trans_menu[TRANS_MENU]={
    {"trans_Wavelet",smeBSBObjectClass,"Wavelet",NULL},
};
MenuItem     comp_menu[COMP_MENU]={
    {"comp_KLICS",smeBSBObjectClass,"KLICS",NULL},
    {"comp_KLICS_SA",smeBSBObjectClass,"KLICS SA",NULL},
};
XtCallbackRec    frame_call[]={
    {FrameDestroy,(caddr_t)frame}, {Free,(caddr_t)sel}, {NULL,NULL},
    {CopyVideo,(caddr_t)video}, {NULL,NULL},
    {Examine,(caddr_t)frame}, {NULL,NULL},
    {FramePointYN,(caddr_t)frame}, {NULL,NULL},
    {FrameInfo,(caddr_t)frame}, {NULL,NULL},
    {FrameMerge,(caddr_t)frame}, {NULL,NULL},
    {Movie,(caddr_t)frame}, {NULL,NULL},
    {PostScript,(caddr_t)frame}, {NULL,NULL},
    {Select,(caddr_t)sel}, {NULL,NULL},
    {Spectrum,(caddr_t)frame}, {NULL,NULL},
}, image_call[]={
    {NewPoint,(caddr_t)frame}, {NULL,NULL},
}, trans_call[]={
    {Transform,(caddr_t)video}, {NULL,NULL},
}, comp_call[]={
    {Compress,(caddr_t)video}, {NULL,NULL},
    {KlicsSA,(caddr_t)video}, {NULL,NULL},
};
Colormap       cmap=ChannelCmap(frame->channel=(video->type==MONO
|| video->trans.type!=TRANS_None)?0:3,video->type,video->gamma);

Dprintf("InitFrame\n");

```

- 177 -

```
sel-> name = "video_Compare";
sel-> button = "frm_compare";
sel-> list_proc = VideoCurrentList;
sel-> action_name = "Compare videos";
sel-> action_proc = Compare;
sel-> action_closure = (caddr_t)video;
frame-> video = video;
frame-> shell = ShellWidget("frm_shell", global-> toplevel, SW_top, cmap, NULL);
form = FormatWidget("frm_form", frame-> shell);
frame-> image_widget = NULL;

frame-> msg = NULL;

frame-> zoom = 0;
frame-> frame = 0;

frame-> point_switch = False;
frame-> point_merge = False;

frame-> point = (Point)MALLOC(sizeof(PointRec));
frame-> point-> location[0] = 0;
frame-> point-> location[1] = 0;
frame-> point-> usage = 1;
frame-> point-> next = global-> points;
global-> points = frame-> point;

frame-> palette = 0;

frame-> next = global-> frames;
global-> frames = frame;

GetFrame(video, frame-> frame);
```

- 178 -

```

pixmap = UpdateImage(frame);

```

```

FillForm(form,FRAME_ICONS,items,widgets,frame_call);
shell[0] = ShellWidget("frm_trans_menu",widgets[4],SW_menu,NULL,NULL);
FillMenu(shell[0],TRANS_MENU,trans_menu,trans_widgets,trans_call);
shell[1] = ShellWidget("frm_comp_menu",widgets[7],SW_menu,NULL,NULL);
FillMenu(shell[1],COMP_MENU,comp_menu,comp_widgets,comp_call);

```

```

frame->point_merge_widget=widgets[6];

```

```

XtSetArg(args[0],XtNbitmap,pixmap);
XtSetArg(args[1],XtNwidth,video->size[0]);
XtSetArg(args[2],XtNheight,video->size[1]);
XtSetArg(args[3],XtNcallback,image_call);

```

```

frame->image_widget = XtCreateManagedWidget("frm_image",imageWidgetClass,widget
s[11],args,FOUR);

```

```

XtSetSensitive(frame->image_widget,False);
XtSetSensitive(widgets[13],PseudoColor==global->visinfo->class);
XtPopup(frame->shell,XtGrabNone);

```

```

}

```

```

Video FindVideo(name,video)

```

```

String name;

```

```

Video video;

```

```

{

```

```

    if (video==NULL) return(NULL);
    else if (!strcmp(name,video->name)) return(video);
    else return(FindVideo(name,video->next));

```

```

}

```

- 179 -

source/InitMain.c

```
/*  
    Initialise menu structure for Main command widget  
*/
```

```
#include    "../include/xwave.h"
```

```
/* Save externs */
```

```
extern void  VideoSave();  
extern void  VideoXimSave();  
extern void  VideoDTSave();  
extern void  VideoMacSave();  
extern void  VideoHexSave();
```

```
/* List externs */
```

```
extern String *VideoList();  
extern String *VideoDropList();  
extern String *VideoCurrentList();  
extern String *KlicsList();  
extern String *KlicsListSA();
```

```
/* Import externs */
```

```
extern void  ImportKlics();  
extern void  ImpKlicsTestSA();
```

```
/* Main externs */
```

- 180 -

```
extern void  Select();
extern void  VideoClean();
extern void  Quit();
extern void  VideoLoad();
extern void  InitFrame();
extern void  VideoDrop();
extern void  PlotGraph();
```

```
/*    Function Name:    InitMain
 *    Description:    Create main menu button & sub-menus
 *    Arguments:    none
 *    Returns:    none
 */
```

```
#define    MAIN_MENU    7
#define    SAVE_MENU    5
#define    IMPT_MENU    2
```

```
InitMain()
```

```
{
    Widget      form=FormatWidget("xwave_form",global->toplevel), widgets[1],
               main_shell, main_widgets[MAIN_MENU],
               save_shell, save_widgets[SAVE_MENU],
               impt_shell, impt_widgets[IMPT_MENU];
    FormItem    items[]={
        {"xwaveLogo","main",0,0,FW_icon_button,"xwave_main_sh"},
    };
    MenuItem    main_menu[]={
        {"main_Open",smeBSBObjectClass,"Open a video",NULL},
        {"main_Attach",smeBSBObjectClass,"Attach a frame",NULL},
        {"main_Save",smeBSBprObjectClass,"Save a video","xwave_save_sh"},
    };
}
```

- 181 -

```

        {"main_Drop", smeBSBObjectClass, "Drop a video", NULL},
        {"main_Clean", smeBSBObjectClass, "Clean out videos", NULL},
        {"main_Import", smeBSBprObjectClass, "Import a
video", "xwave_impt_sh"},
        {"main_Quit", smeBSBObjectClass, "Quit", NULL},
    }, save_menu[] = {
        {"save_menu_vid", smeBSBObjectClass, "Save xwave video", NULL},
        {"save_menu_xim", smeBSBObjectClass, "Save xim video", NULL},
        {"save_menu_dt", smeBSBObjectClass, "Save DT image", NULL},
        {"save_menu_mac", smeBSBObjectClass, "Save mac video", NULL},
        {"save_menu_hex", smeBSBObjectClass, "Save hex dump", NULL},
    }, impt_menu[] = {
        {"impt_menu_klics", smeBSBObjectClass, "KLICS", NULL},
        {"impt_menu_klicsSA", smeBSBObjectClass, "KLICS SA", NULL},
    };
    static SelectItem selection[] = {
        {"video_Open", "xwaveLogo", VideoList, "Open a
video", VideoLoad, NULL},
        {"frame_Attach", "xwaveLogo", VideoCurrentList, "Attach a
frame", InitFrame, NULL},
        {"video_Drop", "xwaveLogo", VideoDropList, "Drop a
video", VideoDrop, NULL},
    }, save_sel[] = {
        {"save_vid", "xwaveLogo", VideoCurrentList, "Save xwave
video", VideoSave, NULL},
        {"save_xim", "xwaveLogo", VideoCurrentList, "Save xim
video", VideoXimSave, NULL},
        {"save_dt", "xwaveLogo", VideoCurrentList, "Save DT
image", VideoDTSave, NULL},
        {"save_mac", "xwaveLogo", VideoCurrentList, "Save mac
video", VideoMacSave, NULL},
        {"save_hex", "xwaveLogo", VideoCurrentList, "Save hex

```


- 182 -

```

dump",VideoHexSave,NULL},
    }, impt_sel[]={
        {"impt_klics","xwaveLogo",KlicsList,"Import
KLICS",ImportKlics,NULL},
        {"impt_klicsSA","xwaveLogo",KlicsListSA,"Import KLICS
SA",ImpKlicsTestSA,NULL},
    };
    XtCallbackRec      main_call[]={
        {Select,(caddr_t)&selection[0]}, {NULL,NULL},
        {Select,(caddr_t)&selection[1]}, {NULL,NULL},
        {Select,(caddr_t)&selection[2]}, {NULL,NULL},
        {VideoClean,(caddr_t)NULL}, {NULL,NULL},
        {Quit,(caddr_t)NULL}, {NULL,NULL},
    }, save_call[]={
        {Select,(caddr_t)&save_sel[0]}, {NULL,NULL},
        {Select,(caddr_t)&save_sel[1]}, {NULL,NULL},
        {Select,(caddr_t)&save_sel[2]}, {NULL,NULL},
        {Select,(caddr_t)&save_sel[3]}, {NULL,NULL},
        {Select,(caddr_t)&save_sel[4]}, {NULL,NULL},
    }, impt_call[]={
        {Select,(caddr_t)&impt_sel[0]}, {NULL,NULL},
        {Select,(caddr_t)&impt_sel[1]}, {NULL,NULL},
    };
    Dprintf("InitMain\n");
    FillForm(form,ONE,items,widgets,NULL);
    main_shell=ShellWidget("xwave_main_sh",widgets[0],SW_menu,NULL,NULL);
    save_shell=ShellWidget("xwave_save_sh",main_shell,SW_menu,NULL,NULL);
    impt_shell=ShellWidget("xwave_impt_sh",main_shell,SW_menu,NULL,NULL);
    FillMenu(main_shell,MAIN_MENU,main_menu,main_widgets,main_call);
    FillMenu(save_shell,SAVE_MENU,save_menu,save_widgets,save_call);
    FillMenu(impt_shell,IMPT_MENU,impt_menu,impt_widgets,impt_call);
}

```

- 183 -

source/Klics5.c

```
/*
    Full still/video Knowles-Lewis Image Compression System utilising HVS
    properties
    and delta-tree coding
*/

#include "xwave.h"
#include "Klics.h"
#include <math.h>

extern Bits bopen();
extern void bclose(), bread(), bwrite(), bflush();

extern WriteKlicsHeader();

/* token modes (empty) */
#define EMPTY 0
#define CHANNEL_EMPTY 1
#define OCTAVE_EMPTY 2
#define LPF_EMPTY 3
#define FULL 4

typedef struct _HistRec {
    int bits, octbits[3][5], lpf, activity, target, token[TOKENS], coeff[129];
    double q_const;
} HistRec, *Hist; /* history record */

/* Function Name: Access
 * Description: Find index address from co-ordinates
```

- 184 -

```

*   Arguments:  x, y - (x,y) co-ordinates
*
*               oct, sub, channel - octave, sub-band and channel co-ordinates
*
*               width - image data width
*
*   Returns:  index into vid->data[channel][][index]
*/

```

```
int   Access(x,y,oct,sub,width)
```

```
int   x, y, oct, sub, width;
```

```

{
    return(((x < < 1)+(sub > > 1)+width*((y < < 1)+(1&sub)))< < oct);
}

```

```

/*   Function Name:      LastFrame
*
*   Description:  Find last frame encoded
*
*   Arguments:  z - index of current frame
*
*               hist - history records
*
*   Returns:      index of previous frame
*/

```

```
int   LastFrame(z,hist)
```

```
int   z;
```

```
Hist hist;
```

```

{
    int   i=z-1;

    while(hist[i].bits==0 && i>0) i--;
    return(i<0?0:i);
}

```

- 185 -

```

/*    Function Name:    Decide
 *    Description:    Calculate value representing the difference between new and old
blocks
 *    Arguments:    new, old - blocks to compare
 *                  mode - differencing algorithm {MAXIMUM | SIGABS |
SIGSQR}
 *    Returns:    difference value
 */

```

```

int    Decide(new,old,mode)

```

```

Block new, old;

```

```

int    mode;

```

```

{
    int    X, Y, sigma=0;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++) {
        int    n_o=new[X][Y]-old[X][Y];

        switch(mode) {
        case MAXIMUM:
            sigma=sigma>abs(n_o)?sigma:abs(n_o);
            break;
        case SIGABS:
            sigma+=abs(n_o);
            break;
        case SIGSQR:
            sigma+=n_o*n_o;
            break;
        }
    }
}

```

- 186 -

```
        return(sigma);
    }

/*  Function Name:    DecideDouble
 *  Description:    Calculates normal w.r.t differencing algorithm
 *  Arguments:    norm - normal value
 *
 *                  mode - differencing algorithm {MAXIMUM | SIGABS |
SIGSQR}
 *  Returns:    new normal value
 */

double    DecideDouble(norm,mode)

double    norm;
int    mode;

{
    double    ret;

    switch(mode) {
    case MAXIMUM:
        ret = norm;
        break;
    case SIGABS:
        ret = 4.0*norm;
        break;
    case SIGSQR:
        ret = 4.0*norm*norm;
        break;
    }
    return(ret);
}
```

- 187 -

Boolean Decision(new,old,norm,mode)

Block new, old;

double norm;

int mode;

```
{
    return((double)Decide(new,old,mode) <= DecideDouble(norm,mode));
}
```

/* Function Name: Feedback

* Description: Calculates new target activity from target bits and historical values

* Arguments: hist - history records

* curr - current frame

* taps - size of history window

* Returns: target activity

*/

int Feedback(hist,curr,taps)

int curr;

Hist hist;

int taps;

```
{
    int      prev=curr, i;
    double      ratio=0;

    for(i=0;i<taps && prev!=0;i++) {
        prev=LastFrame(prev,hist);
```

```
ratio += (double)hist[prev].activity / ((double)(hist[prev].bits - (prev == 0 ? hist[0].lpf : 0)));
```

- 188 -

```

    }
    return((int)(ratio*(double)hist[curr].target/(double)i));
}

```

```

/*  Function Name:    Filter
*   Description:    Calculates new q_const filtering historical values
*   Arguments:    hist - history records
*                  curr - current frame
*                  taps - size of history window
*                  filter - index to filter
*   Returns:      q_const
*/

```

```
double    Filter(hist,curr,taps,filter)
```

```
int    curr;
```

```
Hist    hist;
```

```
int    taps, filter;
```

```

{
    double    mac=hist[curr].q_const, sum=1.0, coeff=1.0;
    int    i, prev=curr;

    for(i=0;i<taps && prev!=0;i++) {
        prev=LastFrame(prev,hist);
        coeff=filter==0?0:coeff/2.0;
        mac+=hist[prev].q_const*coeff;
        sum+=coeff;
    }
    return(mac/sum);
}

```

- 189 -

```
/*  Function Name:    Huffman
 *  Description:    Calculates the number of bits for the Huffman code representing
level
 *  Arguments:    level - level to be encoded
 *  Returns:      number of bits in codeword
 */
```

```
int  Huffman(level)
```

```
int  level;
```

```
{
    return(level == 0?2:(abs(level) < 3?3:1 + abs(level)));
}
```

```
/*  Function Name:    HuffCode
 *  Description:    Generates Huffman code representing level
 *  Arguments:    level - level to be encoded
 *  Returns:      coded bits in char's
 */
```

```
unsigned char *HuffCode(level)
```

```
int  level;
```

```
{
    unsigned char *bytes=(unsigned char *)MALLOC((7+Huffman(level))/8);

    bytes[0]=(abs(level) < 3?abs(level):3) | (level < 0?4:0);
    if (abs(level) > 2) {
        int  index=(7+Huffman(level))/8-1;
```


- 190 -

```
        bytes[index]=bytes[index]|(1 << (Huffman(level)-1)%8);
    }
    return(bytes);
}
```

unsigned char *CodeInt(number,bits)

```
int    number, bits;

{
    int    len=(7+bits)/8;
    unsigned char *bytes=(unsigned char *)MALLOC(len);
    int    byte;

    for(byte=0;byte<len;byte++) {
        bytes[byte]=0xff&number;
        number=number>>8;
    }
    return(bytes);
}
```

int ReadInt(bits,bfp)

```
int    bits;
Bits   bfp;
```

```
{
    int    len=(7+bits)/8;
    unsigned char bytes[len];
    int    byte, number=0;

    bread(bytes,bits,bfp);
```

- 191 -

```

    for(byte=0;byte<len;byte++)
        number=number|((int)bytes[byte]<<byte*8);
    number=(number<<sizeof(int)*8-bits)>>sizeof(int)*8-bits;
    return(number);
}

```

```

/*  Function Name:    HuffRead
 *  Description:    Read Huffman encoded number from binary file
 *  Arguments:    bfp - binary file pointer
 *  Returns:    decoded level
 */

```

```

int    HuffRead(bfp)

```

```

Bits    bfp;

```

```

{
    int    value;
    unsigned char    byte;
    Boolean    negative=False;

    bread(&byte,2,bfp);
    value=(int)byte;
    if (byte=='\0') return(0);
    else {
        bread(&byte,1,bfp);
        negative=(byte!='\0');
    }
    if (value<3) return(negif(negative,value));
    for(byte='\0';byte=='\0';value++) bread(&byte,1,bfp);
    return(negif(negative,value-1));
}

```

- 192 -

```

/*  Function Name:    Quantize
 *  Description:    RM8 style quantizer
 *  Arguments:    data - unquantised number
 *                q - quantizing divisor
 *                level - quantised to level
 *  Returns:      quantized data & level
 */

```

```

int  Quantize(data,q,level)

```

```

int  data, q, *level;

```

```

{
    int  mag_level=abs(data)/q;

    *level=negif(data<0,mag_level);
    return(negif(data<0,mag_level*q+(mag_level!=0?(q-1)>>1:0)));
}

```

```

/*  Function Name:    Proposed
 *  Description:    Calculates proposed block values
 *  Arguments:    pro - proposed block
 *                lev - proposed block quantized levels
 *                old, new - old and new block values
 *                decide - decision algorithm
 *                norms - HVS normals
 *  Returns:      new == 0, proposed values (pro) and levels (lev)
 */

```

```

Boolean  Proposed(pro,lev,old,new,decide,norms)

```

```

Block  pro, lev, old, new;

```

- 193 -

```

int    decide;
double    norms[3];

{
    Block zero_block={{0,0},{0,0}};
    int    X, Y, step=norms[0]<1.0?1:(int)norms[0];
    Boolean    zero=Decision(new,zero_block,norms[1],decide);

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)

pro[X][Y]=zero?0:old[X][Y]+Quantize(new[X][Y]-old[X][Y],step,&(lev[X][Y]));
    return(zero);
}

```

```

/*    Function Name:    ZeroCoeffs
*    Description:    Zero out video data
*    Arguments:    data - image data
*                  addr - addresses
*    Returns:    zeros data[addr[][]]
*/

```

```

void    ZeroCoeffs(data,addr)

```

```

short    *data;
Block    addr;

```

```

{
    int    X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
        data[addr[X][Y]]=0;
}

```

- 194 -

```

/*  Function Name:    BlockZero
 *   Description:    Test if all block values are zero
 *   Arguments:    block - block under test
 *   Returns:      block == 0
 */

```

```

Boolean    BlockZero(block)

```

```

Block block;

```

```

{
    int    X, Y;
    Boolean    zero = True;

    for(X=0; X < BLOCK; X++) for(Y=0; Y < BLOCK; Y++)
        if (block[X][Y] != 0) zero = False;
    return(zero);
}

```

```

/*  Function Name:    SendToken
 *   Description:    Increments token frequency
 *   Arguments:    token - token to be transmitted
 *                  channel, sub, oct - co-ordinates
 *                  ctrl - control record for compression
 *                  hist - history record
 *                  empty - zero state {EMPTY | CHANNEL_EMPTY |
OCTAVE_EMPTY | LPF_EMPTY | FULL}
 *                  branch - branch of tree (0-3)
 *   Returns:      encodes token
 */

```

```

void    SendToken(token, channel, sub, oct, ctrl, hist, empty, branch)

```

- 195 -

```

int    token, channel, sub, oct, *empty, branch;
CompCtrl  ctrl;
Hist    hist;

{
    int    full=FULL, i;
    String
token_name[TOKENS]={ "ZERO_STILL", "NON_ZERO_STILL", "BLOCK_SAME", "ZE
RO_VID", "BLOCK_CHANGE",

"LOCAL_ZERO", "LOCAL_NON_ZERO", "CHANNEL_ZERO", "CHANNEL_NON_ZE
RO", "OCT_ZERO", "OCT_NON_ZERO",

"LPF_ZERO", "LPF_NON_ZERO", "LPF_LOC_ZERO", "LPF_LOC_NON_ZERO"};

    switch(*empty) {
    case EMPTY:
        if (token!=ZERO_STILL && token!=BLOCK_SAME) {

SendToken(LOCAL_NON_ZERO,channel,sub,oct,ctrl,hist,&full,branch);
            for(i=0;i<channel;i++)
SendToken(CHANNEL_ZERO,i,sub,oct,ctrl,hist,&full,branch);
                *empty=CHANNEL_EMPTY;
                SendToken(token,channel,sub,oct,ctrl,hist,empty,branch);
            }
            break;
        case CHANNEL_EMPTY:
            if (token!=ZERO_STILL && token!=BLOCK_SAME) {

SendToken(CHANNEL_NON_ZERO,channel,sub,oct,ctrl,hist,&full,branch);
                for(i=1;i<sub;i++)
SendToken(token==NON_ZERO_STILL?ZERO_STILL:BLOCK_SAME,channel,i,oct,ct

```

- 196 -

```
rl.hist,&full,branch);

    *empty = FULL;
    SendToken(token,channel,sub,oct,ctrl,hist,empty,branch);
}
break;
case OCTAVE_EMPTY:
    if (token!= ZERO_STILL && token!= BLOCK_SAME) {

SendToken(OCT_NON_ZERO,channel,sub,oct,ctrl,hist,&full,branch);
        for(i=0;i < branch;i++)
SendToken(token==NON_ZERO_STILL?ZERO_STILL:BLOCK_SAME,channel,sub,oct,ctrl,hist,&full,branch);
        *empty = FULL;
        SendToken(token,channel,sub,oct,ctrl,hist,empty,branch);
    }
    break;
case LPF_EMPTY:
    if (token!= LPF_ZERO) {

SendToken(LPF_LOC_NON_ZERO,channel,sub,oct,ctrl,hist,&full,branch);
        for(i=0;i < channel;i++)
SendToken(LPF_ZERO,i,sub,oct,ctrl,hist,&full,branch);
        *empty = FULL;
        SendToken(token,channel,sub,oct,ctrl,hist,empty,branch);
    }
    break;
case FULL:
    Dprintf("%s\n",token_name[token]);
    hist->token[token]++;
    hist->bits+=token_bits[token];
    hist->octbits[channel][oct]+=token_bits[token];
    if (ctrl->bin_switch)
```

- 197 -

```

bwrite(&token_codes[token],token_bits[token],ctrl->bfp);
    break;

```

```

    }
}

```

```

/*  Function Name:    ReadBlock
 *  Description:    Read block from video
 *  Arguments:    new, old, addr - new and old blocks and addresses
 *                x, y, z, oct, sub, channel - co-ordinates of block
 *                ctrl - compression control record
 *  Returns:    block values
 */

```

```

void  ReadBlock(new,old,addr,x,y,z,oct,sub,channel,ctrl)

```

```

Block new, old, addr;

```

```

int    x, y, z, oct, sub, channel;

```

```

CompCtrl    ctrl;

```

```

{

```

```

    int    X, Y;

```

```

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++) {

```

```

        addr[X][Y]=Access((x<<1)+X,(y<<1)+Y,oct,sub,Size(ctrl->src,channel,0));

```

```

        new[X][Y]=(int)ctrl->src->data[channel][z][addr[X][Y]];

```

```

        old[X][Y]=(int)ctrl->dst->data[channel][z][addr[X][Y]];

```

```

    }

```

```

}

```

```

/*  Function Name:    CalcNormals

```

```

 *  Description:    Calculates HVS weighted normals

```


- 198 -

```

*   Arguments:  ctrl - compression control record
*
*               oct, sub, channel - co-ordinates
*
*               norms - pre-initialised normals
*
*   Returns:    weighted normals
*/

```

```
void CalcNormals(ctrl,oct,sub,channel,norms)
```

```

CompCtrl  ctrl;
int  oct, sub, channel;
double  norms[3];

{
    Video vid=ctrl->dst;
    int  norm, base_oct=oct+(vid->type==YUV &&
channel!=0?vid->trans.wavelet.space[0]-vid->trans.wavelet.space[1]:0)+(sub==0?1:0)
;

    for(norm=0;norm<3;norm++) {
        if (norm!=0) norms[norm] *= ctrl->quant_const;
        norms[norm] *=
ctrl->base_factors[base_oct]*(sub==3?ctrl->diag_factor:1.0);
        if (channel!=0) norms[norm] *= ctrl->chrome_factor;
        norms[norm] *=(double)(1<<vid->precision);
    }
}

```

```

/*   Function Name:    MakeDecisions
*
*   Description:  Decide on new compression mode from block values
*
*   Arguments:    old, new, pro - block values
*
*               zero - zero flag for new block
*
*               norms - HVS normals

```

- 199 -

```

*           mode - current compression mode
*           decide - comparison algorithm
*   Returns:   new compression mode
*/

```

```
int   MakeDecisions(old,new,pro,zero,norms,mode,decide)
```

```
Block new, old, pro;
```

```
Boolean   zero;
```

```
double    norms[3];
```

```
int   mode, decide;
```

```
{
```

```
Block zero_block={{0,0},{0,0}};
```

```
int   new_mode, np=Decide(new,pro,decide), no=Decide(new,old,decide);
```

```
if (np < no && (double)no > DecideDouble(norms[mode]==STILL?1:2],decide)
&& !zero)
```

```
new_mode=mode==STILL ||
```

```
(double)Decide(old,zero_block,decide) <= DecideDouble(norms[1],decide)?STILL:SEND;
```

```
else new_mode=mode==SEND && np < no && zero?VOID:STOP;
```

```
return(new_mode);
```

```
}
```

```
int   MakeDecisions2(old,new,pro,lev,zero,norms,mode,decide)
```

```
Block new, old, pro, lev;
```

```
Boolean   zero;
```

```
double    norms[3];
```

```
int   mode, decide;
```

```
{
```

- 200 -

```

Block  zero_block = {{0,0},{0,0}};
int    new_mode = mode == STILL || BlockZero(old)?STILL:SEND,
        np = Decide(new,pro,decide), no = Decide(new,old,decide);

    if (new_mode == STILL) new_mode = np > no || zero ||
BlockZero(lev)?STOP:STILL;
    else new_mode = zero && np < no?VOID:np > no ||
Decision(new,old,norms[2],decide) || BlockZero(lev)?STOP:SEND;
    return(new_mode);
}

```

```

/*  Function Name:      UpdateCoeffs
 *
 *  Description:  Encode proposed values and write data
 *
 *  Arguments:   pro, lev, addr - proposed block, levels and addresses
 *
 *               z, channel, oct - co-ordinates
 *
 *               ctrl - compression control record
 *
 *               hist - history record
 *
 *  Returns:     alters ctrl->dst->data[channel][z][addr[]]
 */

```

```
void  UpdateCoeffs(pro,lev,addr,z,channel,oct,ctrl,hist)
```

```

Block  pro, lev, addr;
int    z, channel, oct;
CompCtrl ctrl;
Hist   hist;

```

```

{
    int    X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++) {
        int    bits=Huffman(lev[X][Y]),

```

- 201 -

```

        level = abs(lev[X][Y]);

        ctrl->dst->data[channel][z][addr[X][Y]] = (short)pro[X][Y];
        hist->coeff[level > 128?128:level] ++;
        hist->bits += bits;
        hist->octbits[channel][oct] += bits;
        if (ctrl->bin_switch) {
            unsigned char    *bytes = HuffCode(lev[X][Y]);

            bwrite(bytes,bits,ctrl->bfp);
            XtFree(bytes);
        }
    }
}

/*  Function Name:    SendTree
 *  Description:    Encode tree blocks
 *  Arguments:    prev_mode - compression mode
 *                x, y, z, oct, sub, channel - co-ordinates
 *                ctrl - compression control record
 *                hist - history records
 *                empty - token mode
 *                branch - tree branch number
 *  Returns:    active block indicator
 */

Boolean    SendTree(prev_mode,x,y,z,oct,sub,channel,ctrl,hist,empty,branch)

int    prev_mode, x, y, z, oct, sub, channel, *empty, branch;
CompCtrl    ctrl;
Hist    hist;

```

- 202 -

```

{
    Block addr, old, new, pro, lev;
    int    new_mode, X, Y;
    double

norms[3] = {ctrl->quant_const, ctrl->thresh_const, ctrl->cmp_const}; /* quant, thresh,
compare */
    Boolean    active = False;

    ReadBlock(new, old, addr, x, y, z, oct, sub, channel, ctrl);
    if (prev_mode != VOID) {
        Boolean    zero;

        CalcNormals(ctrl, oct, sub, channel, norms);
        zero = Proposed(pro, lev, old, new, ctrl->decide, norms);
    /*
new_mode = MakeDecisions(old, new, pro, zero, norms, prev_mode, ctrl->decide); */

new_mode = MakeDecisions2(old, new, pro, lev, zero, norms, prev_mode, ctrl->decide);
        switch(new_mode) {
            case STOP:

/*SendToken(prev_mode == STILL?ZERO_STILL:BLOCK_SAME, channel, sub, oct, ctrl, h
ist, empty, branch); */

                SendToken(prev_mode == STILL ||
BlockZero(old)?ZERO_STILL:BLOCK_SAME, channel, sub, oct, ctrl, hist, empty, branch);
                break;
            case STILL:
            case SEND:
                active = True;

/*SendToken(prev_mode == STILL?NON_ZERO_STILL:BLOCK_CHANGE, channel, sub
, oct, ctrl, hist, empty, branch); */

```

- 203 -

```

        SendToken(prev_mode == STILL ||
BlockZero(old)?NON_ZERO_STILL:BLOCK_CHANGE,channel,sub,oct,ctrl,hist,empty,
branch);

        UpdateCoeffs(pro,lev,addr,z,channel,oct,ctrl,hist);
        break;
    case VOID:
        SendToken(ZERO_VID,channel,sub,oct,ctrl,hist,empty,branch);
        ZeroCoeffs(ctrl->dst->data[channel][z],addr);
        break;
    }
} else {
    if (BlockZero(old)) new_mode=STOP;
    else {
        ZeroCoeffs(ctrl->dst->data[channel][z],addr);
        new_mode=VOID;
    }
}
if (oct>0 && new_mode!=STOP) {
    int    mt=OCTAVE_EMPTY, full=FULL;

    Dprintf("x= %d, y= %d, oct= %d sub= %d mode
%d\n",x,y,oct,sub,new_mode);
    for(Y=0;Y<2;Y++) for(X=0;X<2;X++)

(void)SendTree(new_mode,x*2+X,y*2+Y,z,oct-1,sub,channel,ctrl,hist,&mt,X+2*Y);
    if (mt==OCTAVE_EMPTY && new_mode!=VOID)
SendToken(OCT_ZERO,channel,sub,oct,ctrl,hist,&full,0);
}
return(active);
}

/*    Function Name:    SendLPF

```

- 204 -

```

*   Description:  Encode LPF sub-band
*   Arguments:   mode - compression mode
*                   z -   frame number
*                   ctrl - compression control record
*                   hist - history records
*   Returns:     encodes data
*/

```

```
void  SendLPF(mode,z,ctrl,hist)
```

```
CompCtrl  ctrl;
```

```
int  mode, z;
```

```
Hist  hist;
```

```
{
```

```
    Block new, old, pro, lev, addr;
```

```
    int  channel, channels=ctrl->src->type==MONO?1:3, x, y, full=FULL,
        octs_lum=ctrl->src->trans.wavelet.space[0],
```

```
    size[2]={Size(ctrl->src,0,0)>>octs_lum+1,Size(ctrl->src,0,1)>>octs_lum+1};
```

```
    for(y=0;y<size[1];y++) for(x=0;x<size[0];x++) {
```

```
        int  empty=LPF_EMPTY;
```

```
        for(channel=0;channel<channels;channel++) {
```

```
            int  octs=ctrl->src->trans.wavelet.space[ctrl->src->type==YUV
&& channel!=0?1:0],
```

```
                new_mode, X, Y, step, value, bits=0;
```

```
                double
```

```
norms[3]={ctrl->quant_const,ctrl->thresh_const,ctrl->cmp_const};
```

```
        CalcNormals(ctrl,octs-1,0,channel,norms);
```

- 205 -

```

    step = norms[0] < 1.0 ? 1 : (int) norms[0];
    for(bits=0,
value=((1 < < 8+ctrl->dst->precision)-1)/step; value!=0; bits++)
        value = value >> 1;
    ReadBlock(new, old, addr, x, y, z, octs-1, 0, channel, ctrl);

    /* Proposed */
    for(X=0; X < BLOCK; X++) for(Y=0; Y < BLOCK; Y++)

pro[X][Y] = old[X][Y] + Quantize(new[X][Y]-old[X][Y], step, &(lev[X][Y]));

    /* MakeDecisions */

new_mode = mode == STILL ? STILL : Decision(new, old, norms[2], ctrl->decide) ||
BlockZero(lev) ? STOP : SEND;

    switch(new_mode) {
    case SEND:
        SendToken(LPF_NON_ZERO, channel, 0, octs, ctrl, hist, &empty, 0);
        UpdateCocffs(pro, lev, addr, z, channel, octs, ctrl, hist);
        break;
    case STILL:
        for(X=0; X < BLOCK; X++) for(Y=0; Y < BLOCK; Y++) {
            ctrl->dst->data[channel][z][addr[X][Y]] = (short)pro[X][Y];
            hist->bits += bits;
            hist->octbits[channel][octs] += bits;
            if (ctrl->bin_switch) {
                unsigned char *bytes = CodeInt(lev[X][Y], bits);

                bwrite(bytes, bits, ctrl->bfp);
                XtFree(bytes);
            }
        }
    }

```


- 206 -

```

        }
        break;
    case STOP:
        SendToken(LPF_ZERO,channel,0,octs,ctrl,hist,&empty,0);
        break;
    }
}
if (mode!=STILL && empty==LPF_EMPTY)
SendToken(LPF_LOC_ZERO,channel,0,octs_lum,ctrl,hist,&full,0);
}
hist->lpf=hist->bits;
}

```

```

/*  Function Name:    LookAhead
*   Description:    Examine base of tree to calculate new quantizer value
*   Arguments:    z - frame number
*                  ctrl - compression control record
*                  hist - history records
*   Returns:    calculates new ctrl->quant_const
*/

```

```
void LookAhead(z,ctrl,hist)
```

```
CompCtrl    ctrl;
```

```
int    z;
```

```
Hist    hist;
```

```

{
    int    x, y, sub, index, thresh[HISTO], decide=ctrl->decide, act,
           taract=Feedback(hist,z,ctrl->feedback),
           octs=ctrl->src->trans.wavelet.space[0],

```

- 207 -

```

size[2] = {Size(ctrl->src,0,0) > 1+octs, Size(ctrl->src,0,1) > 1+octs};
    Block new, old, addr;
    double      old_quant = ctrl->quant_const;

    ctrl->quant_const = 1.0;
    for(index=0; index < HISTO; index++) thresh[index] = 0;
    for(y=0; y < size[1]; y++) for(x=0; x < size[0]; x++)
for(sub=1; sub < 4; sub++) {
        double      q_thresh[3],
norms[3] = {ctrl->quant_const, ctrl->thresh_const, ctrl->cmp_const};
        Block zero_block = {{0,0},{0,0}};

        ReadBlock(new, old, addr, x, y, z, octs-1, sub, 0, ctrl);
        CalcNormals(ctrl, octs-1, sub, 0, norms);

q_thresh[1] = (double)Decide(new, zero_block, decide)/DecideDouble(norms[1], decide);

q_thresh[2] = (double)Decide(new, old, decide)/DecideDouble(norms[2], decide);
        if (BlockZero(old)) q_thresh[0] = q_thresh[1];
        else q_thresh[0] = q_thresh[2] < q_thresh[1] ? q_thresh[2] : q_thresh[1];
        if (ctrl->decide == SIGSQR) q_thresh[0] = sqrt(q_thresh[0]);

index = (int)((q_thresh[0]-old_quant+HISTO_DELTA)*HISTO/(HISTO_DELTA*2));
        index = index < 0 ? 0 : index > HISTO-1 ? HISTO-1 : index;
        thresh[index]++;
    }
    for(index=HISTO-1, act=0; index >= 0 && act < taract; index--)
act += thresh[index];

ctrl->quant_const = (double)(index+1)*HISTO_DELTA*2.0/HISTO+old_quant-HISTO_
DELTA;

ctrl->quant_const = ctrl->quant_const < 0.0 ? 0.0 : ctrl->quant_const;

```

- 208 -

```

    Dprintf("Target bits %d act %d (real %d) adjust q_const to
%3.2f\n", hist[z].target, taract, act, ctrl->quant_const);
    hist[z].q_const = ctrl->quant_const;
    ctrl->quant_const = Filter(hist, z, ctrl->feedback, ctrl->filter);
    Dprintf("Post filtering q_const to %3.2f\n", ctrl->quant_const);
    if (ctrl->bin_switch) {
        unsigned char *bytes = CodeInt(index + 1 - HISTO/2, HISTO_BITS);

        bwrite(bytes, HISTO_BITS, ctrl->bfp);
        XtFree(bytes);
    }
}

```

```

/*  Function Name:    CompressStats
*   Description:    Compile compression statistics
*   Arguments:    ctrl - compression control record
*                  hist - history records
*   Returns:    plot graphs
*/

```

```
void  CompressStats(ctrl, hist)
```

```
CompCtrl  ctrl;
```

```
Hist  hist;
```

```

{
    FILE  *fp_token, *fp_coeff, *fp_log, *fopen();
    char  file_name[STRLEN];
    int   channel, z, i, sigma;

```

```
sprintf(file_name, "%s%s/ %s.token%s\0", global->home, PLOT_DIR, ctrl->stats_name, P
```

- 209 -

```
LOT_EXT);
```

```
    fp_token = fopen(file_name, "w");
```

```
    sprintf(file_name, "%s%s/%s.coeff%s\0", global->home, PLOT_DIR, ctrl->stats_name, PL
OT_EXT);
```

```
    fp_coeff = fopen(file_name, "w");
```

```
    sprintf(file_name, "%s%s/%s.log%s\0", global->home, PLOT_DIR, ctrl->stats_name, PLO
T_EXT);
```

```
    fp_log = fopen(file_name, "w");
```

```
    fprintf(fp_token, "\"Tokens %s\n", ctrl->name);
```

```
    for(i=0; i < TOKENS; i++) {
```

```
        sigma=0;
```

```
        for(z=0; z < ctrl->src->size[2]; z++) sigma += hist[z].token[i];
```

```
        fprintf(fp_token, "%d %d\n", i, sigma);
```

```
    }
```

```
    fprintf(fp_coeff, "\"Coeffs %s\n", ctrl->name);
```

```
    for(i=0; i < 129; i++) {
```

```
        sigma=0;
```

```
        for(z=0; z < ctrl->src->size[2]; z++) sigma += hist[z].coeff[i];
```

```
        fprintf(fp_coeff, "%d %d\n", i, sigma);
```

```
    }
```

```
    for(i=0; i < 5; i++) {
```

```
        String titles[5] = {"treebits", "activity", "quant", "bits", "ratio"};
```

```
        fprintf(fp_log, "\n\"%s\n", titles[i]);
```

```
        for(z=0; z < ctrl->src->size[2]; z++)
```

```
            switch(i) {
```

```
                case 0: fprintf(fp_log, "%d %d\n", z, hist[z].bits-hist[z].lpf);
```

```
                    break;
```

```
                case 1: fprintf(fp_log, "%d %d\n", z, hist[z].activity);
```

```
                    break;
```

- 210 -

```

        case 2: fprintf(fp_log, "%d %f\n", z.hist[z].q_const);
                break;
        case 3:  fprintf(fp_log, "%d %d\n", z.hist[z].bits);
                break;
        case 4:  fprintf(fp_log, "%d
%f\n", z, (double)(hist[z].bits - (z == 0 ? hist[z].lpf : 0)) / (double)hist[z].activity);
                break;
    }
}

for(channel=0; channel < (ctrl->src->type == MONO ? 1 : 3); channel++) {
    int    octs = ctrl->src->trans.wavelet.space[ctrl->src->type == YUV
&& channel != 0 ? 1 : 0];

    for(i=0; i <= octs; i++) {
        fprintf(fp_log, "\n\channel %d oct %d\n", channel, i);
        for(z=0; z < ctrl->src->size[2]; z++)
            fprintf(fp_log, "%d %d\n", z, hist[z].octbits[channel][i]);
    }
}

fclose(fp_token); fclose(fp_coeff); fclose(fp_log);
}

```

```

/*  Function Name:    CopyFrame
*
*  Description:      Copy frame or zero
*
*  Arguments:        vid - video
*
*                      from, to - source and destination frame numbers
*
*                      zero - zero out flag
*
*  Returns:          alters video->data
*/

```

```
void CopyFrame(vid, from, to, zero)
```

- 211 -

```

Video vid;
int    from, to;
Boolean    zero;

{
    int    i, channel;

    for(channel=0;channel < (vid->type == MONO?1:3);channel++) {
        int    size = Size(vid,channel,0)*Size(vid,channel,1);

        for(i=0;i < size;i++)
            vid->data[channel][to][i] = zero?0:vid->data[channel][from][i];
    }
}

```

```

/*  Function Name:    CompressFrame
 *  Description:    Compress a Frame
 *  Arguments:    ctrl - compression control record
 *                  z - frame number
 *                  hist - history records
 *                  target - target bits
 */

```

```

void    CompressFrame(ctrl,z,hist,target)

```

```

CompCtrl    ctrl;
int    z, target;
Hist    hist;

{
    Video src=ctrl->src, dst=ctrl->dst;
    int    sub, channel, x, y, mode=ctrl->stillvid || z==0?STILL:SEND,

```

- 212 -

```

        octs_lum = src->trans.wavelet.space[0],

size[2] = {Size(src,0,0) > > 1 + octs_lum, Size(src,0,1) > > 1 + octs_lum};

NewFrame(dst,z);
CopyFrame(dst,z-1,z,ctrl->stillvid || z == 0);
GetFrame(src,z);
hist[z].target = target;
if (z != 0 && ctrl->auto_q) LookAhead(z,ctrl,hist);
SendLPF(mode,z,ctrl,&hist[z]);
Dprintf("LPF bits %d\n",hist[z].lpf);
hist[z].q_const = ctrl->quant_const;
for(y=0;y < size[1];y++) for(x=0;x < size[0];x++) {
    int    empty = EMPTY, full = FULL;

    for(channel=0;channel < (dst->type == MONO?1:3);channel++) {
        int    octs = src->trans.wavelet.space[src->type == YUV &&
channel != 0?1:0];

        for(sub=1;sub < 4;sub++) {
            Boolean
active = SendTree(mode,x,y,z,octs-1,sub,channel,ctrl,&hist[z],&empty,0);

            hist[z].activity += channel == 0 && active;
        }
        switch(empty) {
        case FULL:
            empty = CHANNEL_EMPTY;
            break;
        case CHANNEL_EMPTY:
            SendToken(CHANNEL_ZERO,channel,sub,octs-1,ctrl,&hist[z],&full,0);
            break;

```

- 213 -

```

        }
    }
    if (empty == EMPTY)
SendToken(LOCAL_ZERO,channel,sub,octs_lum-1,ctrl,&hist[z],&full,0);
    }
    Dprintf("Activity: %d\n",hist[z].activity);
    FreeFrame(src,z);
}

```

```

/*  Function Name:    SkipFrame
*   Description:     Shuffle frame data as if current frame was skipped
*   Arguments:      vid - video
*                   z - frame number
*   Returns:        alters vid->data
*/

```

```
void  SkipFrame(vid,z)
```

```
Video vid;
```

```
int   z;
```

```

{
    NewFrame(vid,z);
    CopyFrame(vid,z-1,z,False);
    if (z > 1) {
        GetFrame(vid,z-2);
        CopyFrame(vid,z-2,z-1,False);
        FreeFrame(vid,z-2);
    }
}

```

```
/*  Function Name:    CompressCtrl
```


- 214 -

```

*   Description:  Perform KLICS on a video
*   Arguments:   w - Xaw widget
*               closure - compression control record
*               call_data - NULL
*   Returns:     compressed video
*/

```

```
void  CompressCtrl(w,closure,call_data)
```

```
Widget      w;
```

```
caddr_t     closure, call_data;
```

```
{
```

```
    CompCtrl  ctrl=(CompCtrl)closure;
```

```
    int       sigma_bits, frame_count, z, i, buffer=0, frames=ctrl->src->size[2],
```

```
             bpf_in=(64000*ctrl->bitrate)/ctrl->src->rate,
```

```
             bpf_out=(int)(((double)(64000*ctrl->bitrate)/ctrl->fps);
```

```
    FILE      *fopen();
```

```
    char      file_name[STRLEN];
```

```
    HistRec   hist[frames];
```

```
    Message   msg=NewMessage(NULL,60);
```

```
    msg->rows=frames > 10?11:frames+(frames==1?0:1); msg->cols=30;
```

```
    if (global->batch==NULL) {
```

```
        XrCallbackRec  callbacks[]={
```

```
            {CloseMessage,(caddr_t)msg}, {NULL,NULL},
```

```
        };
```

```
    MessageWindow(FindWidget("frm_compress",w),msg,"KLICS",True,callbacks);
```

```
    }
```

```
    Dprintf("CompressCtrl\n");
```

- 215 -

```

    if (ctrl->src->type == YUV &&
        (ctrl->src->trans.wavelet.space[0] != ctrl->src->trans.wavelet.space[1] + ctrl->src->U
        Vsample[0] || ctrl->src->UVsample[0] != ctrl->src->UVsample[1])) {
        Eprintf("Y-UV octaves mis-matched. Check UV-sample");
        return;
    }
    ctrl->dst = CopyHeader(ctrl->src);
    strcpy(ctrl->dst->name, ctrl->name);
    if (ctrl->dst->disk) SaveHeader(ctrl->dst);
    if (ctrl->bin_switch) {

sprintf(file_name, "%s%s/%s%s\0", global->home, KLICS_DIR, ctrl->bin_name, KLICS_
EXT);

        ctrl->bfp = bopen(file_name, "w");
        /* Write some sort of header */
        WriteKlicsHeader(ctrl);
    }
    for(z=0; z < frames; z++) {
        hist[z].bits=0;
        hist[z].lpf=0;
        hist[z].activity=0;
        hist[z].target=0;
        for(i=0; i < 5; i++) hist[z].octbits[0][i]=0;
        for(i=0; i < 5; i++) hist[z].octbits[1][i]=0;
        for(i=0; i < 5; i++) hist[z].octbits[2][i]=0;
        for(i=0; i < TOKENS; i++) hist[z].token[i]=0;
        for(i=0; i < 129; i++) hist[z].coeff[i]=0;
        hist[z].q_const=0.0;
    }
    for(z=0; z < frames; z++) {
        if (z == 0 || !ctrl->buf_switch) {
            CompressFrame(ctrl, z, hist, bpf_out);

```

- 216 -

```

        buffer = 3200 * ctrl-> bitrate + bpf_in;
    } else {
        Boolean      no_skip;

        buffer = bpf_in;
        buffer = buffer < 0 ? 0 : buffer;
        no_skip = buffer < 6400 * ctrl-> bitrate; /* H.261 buffer size */
        if (ctrl-> bin_switch) bwrite(&no_skip, 1, ctrl-> bfp);
        if (no_skip) {
            CompressFrame(ctrl, z, hist, bpf_out/* + bpf_out/2 - buffer*/);
            buffer += hist[z].bits;
        } else SkipFrame(ctrl-> dst, z);
    }
    if (z > 0) {
        SaveFrame(ctrl-> dst, z-1);
        FreeFrame(ctrl-> dst, z-1);
    }
    Mprintf(msg, "%s%03d: %d
bits\n", ctrl-> dst-> name, z + ctrl-> src-> start, hist[z].bits);
    Mflush(msg);
}
SaveFrame(ctrl-> dst, ctrl-> src-> size[2]-1);
FreeFrame(ctrl-> dst, ctrl-> src-> size[2]-1);
if (ctrl-> bin_switch) { bflush(ctrl-> bfp); bclose(ctrl-> bfp); }
if (ctrl-> stats_switch) CompressStats(ctrl, hist);
Dprintf("Compression Complete\n");
sigma_bits = 0, frame_count = 0;
for (z = 0; z < ctrl-> src-> size[2]; z++) {
    sigma_bits += hist[z].bits;
    if (hist[z].bits != 0) frame_count++;
}
if (ctrl-> buf_switch) {

```

- 217 -

```

        Dprintf("Buffer contains %d bits\n",buffer-bpf_in);
        Dprintf("Frame Rate %4.1f
Hz\n",((double)(ctrl->src->rate*(frame_count-1))/((double)(ctrl->src->size[2]-1)));
    }
    if (frames > 1) {
        Mprintf(msg,"Total: %d bits\n",sigma_bits);
        Mflush(msg);
    }
    ctrl->dst->next=global->videos;
    global->videos=ctrl->dst;
}

/*  Function Name:    BatchCompCtrl
 *  Description:    Batch interface to CompressCtrl
 */

void  BatchCompCtrl(w,closure,call_data)

Widget      w;
caddr_t     closure, call_data;

{
    CompCtrl  ctrl=(CompCtrl)closure;

    if (ctrl->src == NULL) ctrl->src = FindVideo(ctrl->src_name,global->videos);
    CompressCtrl(w,closure,call_data);
}

/*  Function Name:    InitCompCtrl
 *  Description:    Initialise the compression control record
 *  Arguments:    name - name of the source video
 *  Returns:      compression control record

```

- 218 -

*/

```
CompCtrl  InitCompCtrl(name)
```

```
String name;
```

```
{
```

```
    CompCtrl  ctrl=(CompCtrl)MALLOC(sizeof(CompCtrlRec));
```

```
    int      i;
```

```
    ctrl->decide=SIGABS;
```

```
    ctrl->feedback=4;
```

```
    ctrl->filter=0;
```

```
    ctrl->stillvid=True;
```

```
    ctrl->stats_switch=False;
```

```
    ctrl->auto_q=True;
```

```
    ctrl->buf_switch=True;
```

```
    ctrl->bin_switch=False;
```

```
    ctrl->cmp_const=0.9;
```

```
    ctrl->thresh_const=0.6;
```

```
    ctrl->quant_const=8.0;
```

```
    ctrl->fps=30.0;
```

```
    ctrl->bitrate=1;
```

```
    for(i=0;i<5;i++) {
```

```
        double      defaults[5]={1.0,0.32,0.16,0.16,0.16};
```

```
        ctrl->base_factors[i]=defaults[i];
```

```
    }
```

```
    ctrl->diag_factor=1.4142136;
```

```
    ctrl->chrome_factor=2.0;
```

```
    strcpy(ctrl->src_name,name);
```

```
    strcpy(ctrl->name,name);
```

- 219 -

```

        strcpy(ctrl->stats_name,name);
        strcpy(ctrl->bin_name,name);
        return(ctrl);
    }

/*    Function Name:    Compress
 *    Description:    X Interface to CompressCtrl
 */

#define    COMP_ICONS    25
#define    VID_ICONS    15

void    Compress(w,closure,call_data)

Widget        w;
caddr_t        closure, call_data;

{
    Video video=(Video)closure;
    CompCtrl    ctrl=InitCompCtrl(video->name);
    int    i, space=video->trans.wavelet.space[0]+1;
    NumInput    num_inputs=(NumInput)MALLOC(2*sizeof(NumInputRec));
    FloatInput    flt_inputs=(FloatInput)MALLOC(6*sizeof(FloatInputRec)),

oct_inputs=(FloatInput)MALLOC(space*sizeof(FloatInputRec));
    Message    msg=NewMessage(ctrl->name,NAME_LEN),
               msg_bin=NewMessage(ctrl->bin_name,NAME_LEN),
               msg_stats=NewMessage(ctrl->stats_name,NAME_LEN);
    XtCallbackRec    destroy_call[]={
        {Free,(caddr_t)ctrl},
        {Free,(caddr_t)num_inputs},
        {Free,(caddr_t)flt_inputs},

```

- 220 -

```

        {Free,(caddr_t)oct_inputs},
        {CloseMessage,(caddr_t)msg},
        {CloseMessage,(caddr_t)msg_bin},
        {CloseMessage,(caddr_t)msg_stats},
        {NULL,NULL},
    };

    Widget      parent=FindWidget("frm_compress",XtParent(w)),
               shell=ShellWidget("klics",parent,SW_below,NULL,destroy_call),
               form=FormatWidget("klics_form",shell),

    dec_shell=ShellWidget("klics_cng_dec",shell,SW_menu,NULL,NULL), dec_widgets[3],

    filt_shell=ShellWidget("klics_cng_filt",shell,SW_menu,NULL,NULL), filt_widgets[2],
               widgets[COMP_ICONS], vid_widgets[VID_ICONS],
    oct_widgets[space*2];

    FormItem    items[]={
        {"klics_cancel","cancel",0,0,FW_icon,NULL},
        {"klics_confirm","confirm",1,0,FW_icon,NULL},
        {"klics_title","Compress a video",2,0,FW_label,NULL},
        {"klics_vid_lab","Video Name:",0,3,FW_label,NULL},
        {"klics_vid",NULL,4,3,FW_text,(String)msg},

        {"klics_stats_lab","Statistics:",0,4,FW_label,NULL},
        {"klics_stats",NULL,4,4,FW_yn,(String)&ctrl->stats_switch},
        {"klics_stats_name",NULL,7,4,FW_text,(String)msg_stats},
        {"klics_bin_lab","KLICS File:",0,6,FW_label,NULL},
        {"klics_bin",NULL,4,6,FW_yn,(String)&ctrl->bin_switch},

        {"klics_bin_name",NULL,10,6,FW_text,(String)msg_bin},
        {"klics_dec_lab","Decision:",0,9,FW_label,NULL},
        {"klics_dec_btn","SigmaAbs",4,9,FW_button,"klics_cng_dec"},
        {"klics_qn_float",NULL,0,12,FW_float,(String)&flt_inputs[0]},
    };

```

- 221 -

```

{"klics_qn_scroll",NULL,4,12,FW_scroll,(String)&flt_inputs[0]},

{"klics_th_float",NULL,0,14,FW_float,(String)&flt_inputs[1]},
{"klics_th_scroll",NULL,4,14,FW_scroll,(String)&flt_inputs[1]},
{"klics_cm_float",NULL,0,16,FW_float,(String)&flt_inputs[2]},
{"klics_cm_scroll",NULL,4,16,FW_scroll,(String)&flt_inputs[2]},
{"klics_ch_float",NULL,0,18,FW_float,(String)&flt_inputs[3]},

{"klics_ch_scroll",NULL,4,18,FW_scroll,(String)&flt_inputs[3]},
{"klics_di_float",NULL,0,20,FW_float,(String)&flt_inputs[4]},
{"klics_di_scroll",NULL,4,20,FW_scroll,(String)&flt_inputs[4]},
{"klics_oct_form",NULL,0,22,FW_form,NULL},
{"klics_vid_form",NULL,0,24,FW_form,NULL},
}, vid_items[]={
{"klics_ic_lab","Image Comp:",0,0,FW_label,NULL},
{"klics_ic",NULL,1,0,FW_yn,(String)&ctrl->stillvid},
{"klics_tg_float",NULL,0,1,FW_float,(String)&flt_inputs[5]},
{"klics_tg_scroll",NULL,1,1,FW_scroll,(String)&flt_inputs[5]},
{"klics_px_int",NULL,0,3,FW_integer,(String)&num_inputs[0]},

{"klics_px_down",NULL,1,3,FW_down,(String)&num_inputs[0]},
{"klics_px_up",NULL,6,3,FW_up,(String)&num_inputs[0]},
{"klics_auto_lab","Auto Quant:",0,5,FW_label,NULL},
{"klics_auto",NULL,1,5,FW_yn,(String)&ctrl->auto_q},
{"klics_buf_lab","Buffer:",0,8,FW_label,NULL},

{"klics_buf",NULL,1,8,FW_yn,(String)&ctrl->buf_switch},
{"klics_buf_bm","None",11,8,FW_button,"klics_cng_filt"},
{"klics_hs_int",NULL,0,10,FW_integer,(String)&num_inputs[1]},
{"klics_hs_down",NULL,1,10,FW_down,(String)&num_inputs[1]},
{"klics_hs_up",NULL,14,10,FW_up,(String)&num_inputs[1]},
}, oct_items[2*space];

```


- 222 -

```

MenuItem    dec_menu[] = {
    {"klics_dec_max", smeBSBObjectClass, "Maximum", NULL},
    {"klics_dec_abs", smeBSBObjectClass, "SigmaAbs", NULL},
    {"klics_dec_sqr", smeBSBObjectClass, "SigmaSqr", NULL},
}, filt_menu[] = {
    {"klics_filt_none", smeBSBObjectClass, "None", NULL},
    {"klics_filt_exp", smeBSBObjectClass, "Exp", NULL},
};

XtCallbackRec    callbacks[] = {
    {Destroy, (caddr_t)shell},
    {NULL, NULL},
    {CompressCtrl, (caddr_t)ctrl},
    {Destroy, (caddr_t)shell},
    {NULL, NULL},
    {ChangeYN, (caddr_t)&ctrl->stats_switch}, {NULL, NULL},
    {ChangeYN, (caddr_t)&ctrl->bin_switch}, {NULL, NULL},
    {FloatIncDec, (caddr_t)&flt_inputs[0]}, {NULL, NULL},
    {FloatIncDec, (caddr_t)&flt_inputs[1]}, {NULL, NULL},
    {FloatIncDec, (caddr_t)&flt_inputs[2]}, {NULL, NULL},
    {FloatIncDec, (caddr_t)&flt_inputs[3]}, {NULL, NULL},
    {FloatIncDec, (caddr_t)&flt_inputs[4]}, {NULL, NULL},
}, vid_call[] = {
    {ChangeYN, (caddr_t)&ctrl->stillvid}, {NULL, NULL},
    {FloatIncDec, (caddr_t)&flt_inputs[5]}, {NULL, NULL},
    {NumIncDec, (caddr_t)&num_inputs[0]}, {NULL, NULL},
    {NumIncDec, (caddr_t)&num_inputs[0]}, {NULL, NULL},
    {ChangeYN, (caddr_t)&ctrl->auto_q}, {NULL, NULL},
    {ChangeYN, (caddr_t)&ctrl->buf_switch}, {NULL, NULL},
    {NumIncDec, (caddr_t)&num_inputs[1]}, {NULL, NULL},
    {NumIncDec, (caddr_t)&num_inputs[1]}, {NULL, NULL},
}, dec_call[] = {
    {SimpleMenu, (caddr_t)&ctrl->decide}, {NULL, NULL},

```

- 223 -

```
        {SimpleMenu,(caddr_t)&ctrl->decide}, {NULL,NULL},
        {SimpleMenu,(caddr_t)&ctrl->decide}, {NULL,NULL},
    }, flt_call[]={
        {SimpleMenu,(caddr_t)&ctrl->filter}, {NULL,NULL},
        {SimpleMenu,(caddr_t)&ctrl->filter}, {NULL,NULL},
    }, oct_call[2*space];
XFontStruct *font;
Arg  args[1];
```

```
msg->rows=1; msg->cols=NAME_LEN;
msg_stats->rows=1; msg_stats->cols=NAME_LEN;
msg_bin->rows=1; msg_bin->cols=NAME_LEN;
ctrl->src=(Video)closure;
```

```
flt_inputs[0].format="Quant: %4.1f";
flt_inputs[0].max=10;
flt_inputs[0].min=0;
flt_inputs[0].value = &ctrl->quant_const;
```

```
flt_inputs[1].format="Thresh: %4.1f";
flt_inputs[1].max=10;
flt_inputs[1].min=0;
flt_inputs[1].value = &ctrl->thresh_const;
```

```
flt_inputs[2].format="Comp: %4.1f";
flt_inputs[2].max=10;
flt_inputs[2].min=0;
flt_inputs[2].value = &ctrl->cmp_const;
```

```
flt_inputs[3].format="Chrome: %4.1f";
flt_inputs[3].max=5;
flt_inputs[3].min=1;
```

- 224 -

```
flt_inputs[3].value = &ctrl->chrome_factor;
```

```
flt_inputs[4].format = "Diag: %4.1f";
```

```
flt_inputs[4].max = 2.0;
```

```
flt_inputs[4].min = 1.0;
```

```
flt_inputs[4].value = &ctrl->diag_factor;
```

```
flt_inputs[5].format = "Target: %4.1f";
```

```
flt_inputs[5].max = 30.0;
```

```
flt_inputs[5].min = 10.0;
```

```
flt_inputs[5].value = &ctrl->fps;
```

```
num_inputs[0].format = "px64k: %1d";
```

```
num_inputs[0].max = 8;
```

```
num_inputs[0].min = 1;
```

```
num_inputs[0].value = &ctrl->bitrate;
```

```
num_inputs[1].format = "History: %1d";
```

```
num_inputs[1].max = 8;
```

```
num_inputs[1].min = 1;
```

```
num_inputs[1].value = &ctrl->feedback;
```

```
for(i=0;i < space;i++) {
```

```
    String format=(char *)MALLOC(20);
```

```
    if (i == 0) sprintf(format, "Octave LPF: %%4.2f");
```

```
    else sprintf(format, "Octave %3d: %%4.2f", space-i-1);
```

```
    oct_inputs[i].format = format;
```

```
    oct_inputs[i].max = 1.0;
```

```
    oct_inputs[i].min = 0.0;
```

```
    oct_inputs[i].value = &ctrl->base_factors[space-i-1];
```

```
    oct_items[2*i].name = "klics_oct_float";
```

- 225 -

```

    oct_items[2*i].contents=NULL;
    oct_items[2*i].fromHoriz=0;
    oct_items[2*i].fromVert=i==0?0:2*i-1;
    oct_items[2*i].type=FW_float;
    oct_items[2*i].hook=(String)&oct_inputs[i];
    oct_items[2*i+1].name="klics_oct_scroll";
    oct_items[2*i+1].contents=NULL;
    oct_items[2*i+1].fromHoriz=1;
    oct_items[2*i+1].fromVert=i==0?0:2*i-1;
    oct_items[2*i+1].type=FW_scroll;
    oct_items[2*i+1].hook=(String)&oct_inputs[i];
    oct_call[2*i].callback=FloatIncDec;
    oct_call[2*i].closure=(String)&oct_inputs[i];
    oct_call[2*i+1].callback=NULL;
    oct_call[2*i+1].closure=NULL;
}

FillForm(form,COMP_ICONS-(video->size[2]>1?0:1),items,widgets,callbacks);
FillForm(widgets[23],2*space,oct_items,oct_widgets,oct_call);
FillMenu(dec_shell,THREE,dec_menu,dec_widgets,dec_call);
font=FindFont(widgets[12]);

XtSetArg(args[0],XtNwidth,2+TextWidth(0,"Maximum\nSigmaAbs\nSigmaSqr",font));
XtSetValues(widgets[12],args,ONE);
if (video->size[2]>1) {
    FillForm(widgets[24],VID_ICONS,vid_items,vid_widgets,vid_call);
    FillMenu(filt_shell,TWO,filt_menu,filt_widgets,filt_call);
    font=FindFont(vid_widgets[11]);
    XtSetArg(args[0],XtNwidth,2+TextWidth(0,"None\nExp",font));
    XtSetValues(vid_widgets[11],args,ONE);
}
XtPopup(shell,XtGrabExclusive);
}

```

- 226 -

source/KlicsSA.c

```
/*
    Full still/video Knowles-Lewis Image Compression System utilising HVS
    properties
    and delta-tree coding

    Stand-Alone version uses fixed image format and static data structures
*/

#include    "KlicsSA.h"
#include    <math.h>

extern void  Convolve();

/* useful X definitions */
typedef    char  Boolean;
#define True  1
#define False 0
#define      String char*

/* token modes (empty) */
#define      EMPTY      0
#define      CHANNEL_EMPTY  1
#define      OCTAVE_EMPTY  2
#define      LPF_EMPTY     3
#define      FULL         4

/*  Function Name:      AccessSA
 *  Description:  Find index address from co-ordinates
 *  Arguments:    x, y - (x,y) co-ordinates
```

- 227 -

```

*           oct, sub, channel - octave, sub-band and channel co-ordinates
*   Returns: index into data[channel][index]
*/

```

```
int   AccessSA(x,y,oct,sub,channel)
```

```
int   x, y, oct, sub, channel;
```

```
{
```

```

return((((x < < 1) + (sub > > 1) + (SA_WIDTH > > (channel == 0?0:1)) * ((y < < 1) + (1 & sub)
)) < < oct);
}

```

```
/*   Function Name:   DecideSA
```

```

*   Description:   Calculate value representing the difference between new and old
blocks

```

```
*   Arguments:   new, old - blocks to compare
```

```
*   Returns:     difference value
```

```
*/
```

```
int   DecideSA(new,old)
```

```
Block new, old;
```

```
{
```

```
    int   X, Y, sigma=0;
```

```
        for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
```

```
sigma += abs(new[X][Y]-old[X][Y]);
```

```
    return(sigma);
```

```
}
```

- 228 -

```
/*  Function Name:    DecideDoubleSA
 *  Description:    Calculates normal w.r.t differencing algorithm
 *  Arguments:    norm - normal value
 *  Returns:    new normal value
 */
```

```
double    DecideDoubleSA(norm)
```

```
double    norm;
```

```
{
    return(4.0*norm);
}
```

```
Boolean    DecisionSA(new,old,norm)
```

```
Block new, old;
```

```
double    norm;
```

```
{
    return((double)DecideSA(new,old) <= DecideDoubleSA(norm));
}
```

```
/*  Function Name:    HuffmanSA
```

```
 *  Description:    Calculates the number of bits for the Huffman code representing
level
```

```
 *  Arguments:    level - level to be encoded
 *  Returns:    number of bits in codeword
 */
```

```
int    HuffmanSA(level)
```

- 229 -

```
int    level;
```

```
{
    return(level == 0?2:(abs(level) < 3?3:1 + abs(level)));
}
```

```
/*    Function Name:    HuffCodeSA
 *    Description:    Generates Huffman code representing level
 *    Arguments:    level - level to be encoded
 *    Returns:    coded bits in char's
 */
```

```
unsigned char *HuffCodeSA(level)
```

```
int    level;
```

```
{
    unsigned char *bytes=(unsigned char *)MALLOC((7+Huffman(level))/8);

    bytes[0]=(abs(level) < 3?abs(level):3)|(level < 0?4:0);
    if (abs(level) > 2) {
        int    index=(7+Huffman(level))/8-1;

        bytes[index]=bytes[index]|(1 << (Huffman(level)-1)%8);
    }
    return(bytes);
}
```

```
unsigned char *CodeIntSA(number,bits)
```

```
int    number, bits;
```


- 230 -

```

{
    int    len=(7+bits)/8;
    unsigned char *bytes=(unsigned char *)MALLOC(len);
    int    byte;

    for(byte=0;byte<len;byte++) {
        bytes[byte]=0xff&number;
        number=number>>8;
    }
    return(bytes);
}

int    ReadIntSA(bits,bfp)

int    bits;
Bits   bfp;

{
    int    len=(7+bits)/8;
    unsigned char bytes[len];
    int    byte, number=0;

    bread(bytes,bits,bfp);
    for(byte=0;byte<len;byte++)
        number=number|((int)bytes[byte]<<byte*8);
    number=(number<<sizeof(int)*8-bits)>>sizeof(int)*8-bits;
    return(number);
}

/*  Function Name:    HuffReadSA
*   Description:    Read Huffman encoded number from binary file
*   Arguments:    bfp - binary file pointer

```

- 231 -

```

*   Returns:      decoded level
*/

int   HuffReadSA(bfp)

Bits  bfp;

{
    int    value;
    unsigned char    byte;
    Boolean    negative = False;

    bread(&byte, 2, bfp);
    value = (int)byte;
    if (byte == '\0') return(0);
    else {
        bread(&byte, 1, bfp);
        negative = (byte != '\0');
    }
    if (value < 3) return(negif(negative, value));
    for(byte = '\0'; byte == '\0'; value++) bread(&byte, 1, bfp);
    return(negif(negative, value-1));
}

/*   Function Name:      QuantizeSA
*   Description:  RM8 style quantizer
*   Arguments:   data - unquantised number
*                q - quantizing divisor
*                level - quantised to level
*   Returns:      quantized data & level
*/

```

- 232 -

```

int    QuantizeSA(data,q,level)

int    data, q, *level;

{
    int    mag_level=abs(data)/q;

    *level=negif(data<0,mag_level);
    return(negif(data<0,mag_level*q+(mag_level!=0?(q-1)>>1:0)));
}

```

```

/*  Function Name:    ProposedSA
 *   Description:    Calculates proposed block values
 *   Arguments:    pro - proposed block
 *                  lev - proposed block quantized levels
 *                  old, new - old and new block values
 *                  norms - HVS normals
 *   Returns:    new==0, proposed values (pro) and levels (lev)
 */

```

```

Boolean    ProposedSA(pro,lev,old,new,norms)

```

```

Block    pro, lev, old, new;

```

```

double    norms[3];

```

```

{
    Block    zero_block={{0,0},{0,0}};
    int    X, Y, step=norms[0]<1.0?1:(int)norms[0];
    Boolean    zero=DecisionSA(new,zero_block,norms[1]);

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)

```

- 233 -

```

pro[X][Y]=zero?0:old[X][Y]+Quantize(new[X][Y]-old[X][Y],step,&(lev[X][Y]));
    return(zero);
}

```

```

/*  Function Name:    ZeroCoeffsSA
 *   Description:    Zero out video data
 *   Arguments:    data - image data
 *                  addr - addresses
 *   Returns:      zeros data[addr[]]
 */

```

```
void  ZeroCoeffsSA(data,addr)
```

```

short *data;
Block addr;

```

```

{
    int    X, Y;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
        data[addr[X][Y]]=0;
}

```

```

/*  Function Name:    BlockZeroSA
 *   Description:    Test if all block values are zero
 *   Arguments:    block - block under test
 *   Returns:      block==0
 */

```

```
Boolean  BlockZeroSA(block)
```

```
Block block;
```

- 234 -

```

{
    int    X, Y;
    Boolean    zero=True;

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)
        if (block[X][Y]!=0) zero=False;
    return(zero);
}

```

```

/*  Function Name:    SendTokenSA
    *  Description:    Increments token frequency
    *  Arguments:    token - token to be transmitted
    *                  channel, sub, oct - co-ordinates
    *                  bfp - binary file pointer
    *                  empty - zero state {EMPTY | CHANNEL_EMPTY |
OCTAVE_EMPTY | LPF_EMPTY | FULL}
    *                  branch - branch of tree (0-3)
    *  Returns:        encodes token
    */

```

```

void    SendTokenSA(token,channel,sub,oct,bfp,empty,branch)

```

```

int    token, channel, sub, oct, *empty, branch;
Bits    bfp;

```

```

{
    int    full=FULL, i;
    String
token_name[TOKENS]={ "ZERO_STILL", "NON_ZERO_STILL", "BLOCK_SAME", "ZE
RO_VID", "BLOCK_CHANGE",

"LOCAL_ZERO", "LOCAL_NON_ZERO", "CHANNEL_ZERO", "CHANNEL_NON_ZE

```

- 235 -

RO", "OCT_ZERO", "OCT_NON_ZERO",

"LPF_ZERO", "LPF_NON_ZERO", "LPF_LOC_ZERO", "LPF_LOC_NON_ZERO"};

switch(*empty) {

case EMPTY:

if (token! = ZERO_STILL && token! = BLOCK_SAME) {

SendTokenSA(LOCAL_NON_ZERO, channel, sub, oct, bfp, &full, branch);

for(i=0; i < channel; i++)

SendTokenSA(CHANNEL_ZERO, i, sub, oct, bfp, &full, branch);

*empty = CHANNEL_EMPTY;

SendTokenSA(token, channel, sub, oct, bfp, empty, branch);

}

break;

case CHANNEL_EMPTY:

if (token! = ZERO_STILL && token! = BLOCK_SAME) {

SendTokenSA(CHANNEL_NON_ZERO, channel, sub, oct, bfp, &full, branch);

for(i=1; i < sub; i++)

SendTokenSA(token == NON_ZERO_STILL ? ZERO_STILL : BLOCK_SAME, channel, i, oct,
t, bfp, &full, branch);

*empty = FULL;

SendTokenSA(token, channel, sub, oct, bfp, empty, branch);

}

break;

case OCTAVE_EMPTY:

if (token! = ZERO_STILL && token! = BLOCK_SAME) {

SendTokenSA(OCT_NON_ZERO, channel, sub, oct, bfp, &full, branch);

for(i=0; i < branch; i++)

SendTokenSA(token == NON_ZERO_STILL ? ZERO_STILL : BLOCK_SAME, channel, sub

- 236 -

```

.oct,bfp,&full.branch);
        *empty = FULL;
        SendTokenSA(token,channel,sub,oct,bfp,empty,branch);
    }
    break;
case LPF_EMPTY:
    if (token != LPF_ZERO) {

SendTokenSA(LPF_LOC_NON_ZERO,channel,sub,oct,bfp,&full,branch);
        for(i=0;i < channel;i++)
SendTokenSA(LPF_ZERO,i,sub,oct,bfp,&full,branch);
        *empty = FULL;
        SendTokenSA(token,channel,sub,oct,bfp,empty,branch);
    }
    break;
case FULL:
    Dprintf("%s\n",token_name[token]);
    bwrite(&token_codes[token],token_bits[token],bfp);
    break;
}
}

```

```

/*  Function Name:    ReadBlockSA
*
*  Description:    Read block from video
*
*  Arguments:    new, old, addr - new and old blocks and addresses
*
*                x, y, oct, sub, channel - co-ordinates of block
*
*                src, dst - frame data
*
*  Returns:    block values
*/

```

```

void  ReadBlockSA(new,old,addr,x,y,oct,sub,channel,src,dst)

```

- 237 -

```
Block new, old, addr;
```

```
int    x, y, oct, sub, channel;
```

```
short *src[3], *dst[3];
```

```
{
```

```
    int    X, Y;
```

```
    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++) {
```

```
        addr[X][Y]=AccessSA((x<<1)+X,(y<<1)+Y,oct,sub,channel);
```

```
        new[X][Y]=(int)src[channel][addr[X][Y]];
```

```
        old[X][Y]=(int)dst[channel][addr[X][Y]];
    }
```

```
}
```

```
/*    Function Name:    CalcNormalsSA
```

```
 *    Description:    Calculates HVS weighted normals
```

```
 *    Arguments:    oct, sub, channel - co-ordinates
```

```
 *                  norms - pre-initialised normals
```

```
 *    Returns:    weighted normals
```

```
*/
```

```
void    CalcNormalsSA(oct,sub,channel,norms,quant_const)
```

```
int    oct, sub, channel;
```

```
double    norms[3], quant_const;
```

```
{
```

```
    int    norm, base_oct=oct+(channel!=0?1:0)+(sub==0?1:0);
```

```
    for(norm=0;norm<3;norm++) {
```

```
        if (norm!=0) norms[norm] *= quant_const;
```

```
        norms[norm] *= base_factors[base_oct]*(sub==3?diag_factor:1.0);
    }
```


- 238 -

```

        if (channel!=0) norms[norm] *= chrome_factor;
        norms[norm] *= (double)(1 < < SA_PRECISION);
    }
}

/*  Function Name:    MakeDecisions2SA
 *  Description:    Decide on new compression mode from block values
 *  Arguments:    old, new, pro - block values
 *
 *                zero - zero flag for new block
 *                norms - HVS normals
 *                mode - current compression mode
 *                decide - comparison algorithm
 *  Returns:    new compression mode
 */

int  MakeDecisions2SA(old,new,pro,lev,zero,norms,mode)

Block new, old, pro, lev;
Boolean    zero;
double    norms[3];
int    mode;

{
    Block zero_block={{0,0},{0,0}};
    int    new_mode=mode==STILL || BlockZeroSA(old)?STILL:SEND,
           np=DecideSA(new,pro), no=DecideSA(new,old);

    if (new_mode==STILL) new_mode=np>=no || zero ||
BlockZeroSA(lev)?STOP:STILL;
    else new_mode=zero && np<no?VOID:np>=no ||
DecisionSA(new,old,norms[2]) || BlockZeroSA(lev)?STOP:SEND;
    return(new_mode);
}

```

- 239 -

}

```

/*  Function Name:      UpdateCoeffsSA
 *
 *  Description:  Encode proposed values and write data
 *
 *  Arguments:   pro, lev, addr - proposed block, levels and addresses
 *
 *               channel, oct - co-ordinates
 *
 *               dst - destination data
 *
 *               bfp - binary file pointer
 *
 *  Returns:     alters dst[channel][addr[]]
 */

```

```

void  UpdateCoeffsSA(pro,lev,addr,channel,oct,dst,bfp)

```

```

Block  pro, lev, addr;

```

```

int    channel, oct;

```

```

short  *dst[3];

```

```

Bits   bfp;

```

{

```

    int    X, Y;

```

```

    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++) {

```

```

        int    bits=HuffmanSA(lev[X][Y]),

```

```

            level=abs(lev[X][Y]);

```

```

        unsigned char    *bytes=HuffCodeSA(lev[X][Y]);

```

```

        dst[channel][addr[X][Y]]=(short)pro[X][Y];

```

```

        bwrite(bytes,bits,bfp);

```

```

        XtFree(bytes);

```

```

    }

```

}

- 240 -

```

/*  Function Name:      SendTreeSA
 *
 *  Description:      Encode tree blocks
 *
 *  Arguments:      prev_mode - compression mode
 *
 *                  x, y, oct, sub, channel - co-ordinates
 *
 *                  empty - token mode
 *
 *                  branch - tree branch number
 *
 *  Returns:      active block indicator
 */

```

Boolean

SendTreeSA(prev_mode.x.y.oct.sub.channel.src.dst.empty,branch.quant_const.bfp)

```

int      prev_mode, x, y, oct, sub, channel, *empty, branch;
short    *src[3], *dst[3];
double    quant_const;
Bits      bfp;

{
    Block addr, old, new, pro, lev;
    int     new_mode, X, Y;
    double   norms[3]={quant_const.thresh_const.cmp_const}; /* quant. thresh.
compare */
    Boolean   active = False;

    ReadBlockSA(new,old,addr.x.y.oct.sub.channel.src.dst);
    if (prev_mode!=VOID) {
        Boolean      zero;

        CalcNormalsSA(oct.sub.channel.norms.quant_const);
        zero=ProposedSA(pro,lev,old,new,norms);
        new_mode=MakeDecisions2SA(old,new,pro,lev,zero,norms,prev_mode);
        switch(new_mode) {

```

- 241 -

```

    case STOP:
        SendTokenSA(prev_mode == STILL ||
BlockZeroSA(old)?ZERO_STILL:BLOCK_SAME.channel.sub.oct.bfp.empty,branch);
        break;
    case STILL:
    case SEND:
        active = True;
        SendTokenSA(prev_mode == STILL ||
BlockZero(old)?NON_ZERO_STILL:BLOCK_CHANGE.channel.sub.oct.bfp.empty,bran
ch);

        UpdateCoeffsSA(pro,lev,addr,channel.oct.dst.bfp);
        break;
    case VOID:
        SendTokenSA(ZERO_VID,channel.sub.oct.bfp.empty,branch);
        ZeroCoeffsSA(dst[channel],addr);
        break;
    }
} else {
    if (BlockZeroSA(old)) new_mode = STOP;
    else {
        ZeroCoeffsSA(dst[channel],addr);
        new_mode = VOID;
    }
}

if (oct > 0 && new_mode != STOP) {
    int    mt = OCTAVE_EMPTY, full = FULL;

    Dprintf("x = %d, y = %d, oct = %d sub = %d mode
%d\n".x,y,oct.sub.new_mode);

    for(Y=0;Y<2;Y++) for(X=0;X<2;X++)

(void)SendTreeSA(new_mode.x*2+X,y*2+Y,oct-1.sub.channel.src.dst.&mt.X+2*Y.qua

```

- 242 -

```

nt_const.bfp);
    if (mt == OCTAVE_EMPTY && new_mode != VOID)
SendTokenSA(OCT_ZERO.channel.sub.oct.bfp,&full,0);
    }
    return(active);
}

```

```

/*  Function Name:    SendLPF_SA
 *  Description:    Encode LPF sub-band
 *  Arguments:    mode - compression mode
 *  Returns:    encodes data
 */

```

```

void  SendLPF_SA(mode,src,dst,bfp,quant_const)

```

```

int    mode;
short  *src[3], *dst[3];
Bits   bfp;
double  quant_const;

```

```

{
    Block new, old, pro, lev, addr;
    int    channel, channels=3, x, y, full=FULL,
           octs_lum=3,

size[2]={SA_WIDTH>>octs_lum+1,SA_HEIGHT>>octs_lum+1};

    for(y=0;y<size[1];y++) for(x=0;x<size[0];x++) {
        int    empty=LPF_EMPTY;

        for(channel=0;channel<channels;channel++) {
            int    octs=channel!=0?2:3,

```

- 243 -

```

        new_mode, X, Y, step, value, bits=0;
double      norms[3]={quant_const.thresh_const.cmp_const};

CalcNormalsSA(octs-1,0,channel.norms.quant_const);
step=norms[0]<1.0?1:(int)norms[0];
for(bits=0, value=((1<<8+SA_PRECISION)-1)/step;value!=0;bits++)
    value=value>>1;
ReadBlockSA(new,old,addr,x,y,octs-1,0,channel.src,dst);

/* Proposed */
for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++)

pro[X][Y]=old[X][Y]+QuantizeSA(new[X][Y]-old[X][Y],step,&(lev[X][Y]));

/* MakeDecisions */
new_mode=mode==STILL?STILL:DecisionSA(new,old,norms[2]) ||
BlockZeroSA(lev)?STOP:SEND;

switch(new_mode) {
case SEND:
    SendTokenSA(LPF_NON_ZERO,channel,0,octs,bfp,&empty,0);
    UpdateCoeffsSA(pro,lev,addr,channel,octs,dst,bfp);
    break;
case STILL:
    for(X=0;X<BLOCK;X++) for(Y=0;Y<BLOCK;Y++) {
        unsigned char *bytes=CodeIntSA(lev[X][Y],bits);

        dst[channel][addr[X][Y]]=(short)pro[X][Y];
        bwrite(bytes,bits,bfp);
        XtFree(bytes);
    }
    break;
}

```

- 244 -

```

    case STOP:
        SendTokenSA(LPF_ZERO.channel.0.octs.bfp,&empty,0);
        break;
    }
}

if (mode!=STILL && empty==LPF_EMPTY)
SendTokenSA(LPF_LOC_ZERO.channel.0.octs_lum.bfp,&full,0);
}
}

```

```

/*  Function Name:    CompressFrameSA
 *  Description:      Compress a Frame
 *  Arguments:        mode - compression mode STILL or SEND
 *                    src, dst - source and destination data
 *                    bfp - binary file pointer for result
 *                    quant_const - quantization parameter
 */

```

```
void CompressFrameSA(mode,src,dst,bfp,quant_const)
```

```
int mode;
```

```
short *src[3], *dst[3];
```

```
Bits bfp;
```

```
double quant_const;
```

```
{
```

```
    int sub, channel, x, y, i,
```

```
        octs_lum=3,
```

```
size[2]={SA_WIDTH>>1+octs_lum.SA_HEIGHT>>1+octs_lum};
```

```
for(channel=0;channel<3;channel++) {
```

- 245 -

```

        int
frame_size[2] = {SA_WIDTH > > (channel == 0 ? 0 : 1), SA_HEIGHT > > (channel == 0 ? 0 : 1
)}).

        frame_area = frame_size[0] * frame_size[1];

        for(i=0; i < frame_area; i++)
src[channel][i] = src[channel][i] << SA_PRECISION;
        Convolve(src[channel], False, frame_size, 0, channel == 0 ? 3 : 2);
    }
    bwrite((char *)&quant_const, sizeof(double)*8, bfp);
    SendLPF_SA(mode, src, dst, bfp, quant_const);
    for(y=0; y < size[1]; y++) for(x=0; x < size[0]; x++) {
        int    empty = EMPTY, full = FULL;

        for(channel=0; channel < 3; channel++) {
            int    octs = channel != 0 ? 2 : 3;

            for(sub=1; sub < 4; sub++)
(void)SendTreeSA(mode, x, y, octs-1, sub, channel, src, dst, &empty, 0, quant_const, bfp);
            switch(empty) {
                case FULL:
                    empty = CHANNEL_EMPTY;
                    break;
                case CHANNEL_EMPTY:
SendTokenSA(CHANNEL_ZERO, channel, sub, octs-1, bfp, &full, 0);
                    break;
            }
        }
        if (empty == EMPTY)
SendTokenSA(LOCAL_ZERO, channel, sub, octs_lum-1, bfp, &full, 0);
    }
}

```


- 246 -

source/KlicsTestSA.c

```
#include "xwave.h"
#include "KlicsSA.h"
```

```
extern void CompressFrameSA();
```

```
typedef struct {
    Video src;
    char bin_name[STRLEN];
    Boolean stillvid;
    double quant_const;
} KlicsCtrlRec, *KlicsCtrl;
```

```
/* Function Name: KlicsCtrlSA
 * Description: Test harness for KlicsSA in xwave
 * Arguments: w - Xaw widget
 *             closure - compression control record
 *             call_data - NULL
 * Returns: send data to binary file
 */
```

```
void KlicsCtrlSA(w, closure, call_data)
```

```
Widget w;
caddr_t closure, call_data;
```

```
{
    KlicsCtrl ctrl=(KlicsCtrl)closure;
    int sizeY=SA_WIDTH*SA_HEIGHT,
```

- 247 -

```

        sizeUV = SA_WIDTH*SA_HEIGHT/4, i, z;

short  *dst[3]={
    (short *)MALLOC(sizeof(short)*sizeY),
    (short *)MALLOC(sizeof(short)*sizeUV),
    (short *)MALLOC(sizeof(short)*sizeUV),
}, *src[3]={
    (short *)MALLOC(sizeof(short)*sizeY),
    (short *)MALLOC(sizeof(short)*sizeUV),
    (short *)MALLOC(sizeof(short)*sizeUV),
};

char   file_name[STRLEN];
Bits   bfp;
Boolean   true = True, false = False;

for(i=0;i<sizeY;i++) dst[0][i]=0;
for(i=0;i<sizeUV;i++) { dst[1][i]=0; dst[2][i]=0; }

sprintf(file_name,"%s%s/%s%s\0",global->home,KLICCS_SA_DIR,ctrl->bin_name,KLI
CS_SA_EXT);
bfp=bopen(file_name,"w");
bwrite(&ctrl->stillvid,1,bfp);
bwrite(&ctrl->src->size[2],sizeof(int)*8,bfp);
for(z=0;z<ctrl->src->size[2];z++) {
    GetFrame(ctrl->src,z);
    for(i=0;i<sizeY;i++) src[0][i]=ctrl->src->data[0][z][i];
    for(i=0;i<sizeUV;i++) {
        src[1][i]=ctrl->src->data[1][z][i];
        src[2][i]=ctrl->src->data[2][z][i];
    }
    CompressFrameSA(z==0 ||

```

- 248 -

```
ctrl->stillvid?STILL:SEND.src.dst.bfp.ctrl->quant_const);
```

```
    FreeFrame(ctrl->src.z);
```

```
    }
```

```
    bflush(bfp);
```

```
    bclose(bfp);
```

```
    XtFree(dst[0]);
```

```
    XtFree(dst[1]);
```

```
    XtFree(dst[2]);
```

```
    XtFree(src[0]);
```

```
    XtFree(src[1]);
```

```
    XtFree(src[2]);
```

```
}
```

```
KlicsCtrl    InitKlicsCtrl(name)
```

```
String name;
```

```
{
```

```
    KlicsCtrl    ctrl=(KlicsCtrl)MALLOC(sizeof(KlicsCtrlRec));
```

```
    ctrl->stillvid = True;
```

```
    ctrl->quant_const = 8.0;
```

```
    strcpy(ctrl->bin_name.name);
```

```
    return(ctrl);
```

```
}
```

```
#define    KLICS_SA_ICONS 8
```

```
#define KLICS_SA_VID_ICONS 2
```

```
void    KlicsSA(w.closure.call_data)
```

```
Widget    w;
```

- 249 -

```

caddr_t      closure, call_data;

{
    Video video=(Video)closure;
    KlicsCtrl  ctrl=InitKlicsCtrl(video->name);
    FloatInput flt_inputs=(FloatInput)MALLOC(sizeof(FloatInputRec));
    Message    msg_bin=NewMessage(ctrl->bin_name,NAME_LEN);
    XtCallbackRec  destroy_call[]={
        {Free,(caddr_t)ctrl},
        {Free,(caddr_t)flt_inputs},
        {CloseMessage,(caddr_t)msg_bin},
        {NULL,NULL},
    };
    Widget      parent=FindWidget("frm_compress",XtParent(w)),

    shell=ShellWidget("klicsSA",parent,SW_below,NULL,destroy_call),
        form=FormatWidget("klicsSA_form",shell),
        widgets[KLICS_SA_ICONS],
    vid_widgets[KLICS_SA_VID_ICONS];
    FormItem    items[]={
        {"klicsSA_cancel","cancel",0,0,FW_icon,NULL},
        {"klicsSA_confirm","confirm",1,0,FW_icon,NULL},
        {"klicsSA_title","Run Klics SA",2,0,FW_label,NULL},
        {"klicsSA_bin_lab","KLICS File:",0,3,FW_label,NULL},
        {"klicsSA_bin_name".NULL,4,3,FW_text,(String)msg_bin},

        {"klicsSA_qn_float".NULL,0,5,FW_float,(String)&flt_inputs[0]},
        {"klicsSA_qn_scroll".NULL,6,5,FW_scroll,(String)&flt_inputs[0]},
        {"klicsSA_vid_form".NULL,0,7,FW_form,NULL},
    }, vid_items[]={
        {"klicsSA_ic_lab"."Image Comp:",0,0,FW_label,NULL},
        {"klicsSA_ic".NULL,1,0,FW_yn,(String)&ctrl->stillvid},
    };
};

```

- 250 -

```

XtCallbackRec    callbacks[]={
    {Destroy,(caddr_t)shell},
    {NULL,NULL},
    {KlicsCtrlSA,(caddr_t)ctrl},
    {Destroy,(caddr_t)shell},
    {NULL,NULL},
    {FloatIncDec,(caddr_t)&flt_inputs[0]}, {NULL,NULL},
}, vid_call[]={
    {ChangeYN,(caddr_t)&ctrl->stillvid}, {NULL,NULL},
};

```

```
ctrl->src=video;
```

```
msg_bin->rows=1; msg_bin->cols=NAME_LEN;
```

```
flt_inputs[0].format="Quant: %4.1f";
```

```
flt_inputs[0].max=10;
```

```
flt_inputs[0].min=0;
```

```
flt_inputs[0].value= &ctrl->quant_const;
```

```
FillForm(form,KLICS_SA_ICONS-(video->size[2]>1?0:1),items.widgets,callbacks);
```

```
if (video->size[2]>1)
```

```
FillForm(widgets[7],KLICS_SA_VID_ICONS,vid_items,vid_widgets,vid_call);
```

```
XtPopup(shell,XtGrabExclusive);
```

```
}
```

- 251 -

source/Malloc.c

```
/*
    Memory allocation routine
*/

#include    <stdio.h>

char  *MALLOC(size)

int    size;

{
    char *ptr=(char *)calloc(1,size);

    if (ptr == NULL) Eprintf("Unable to allocate %d bytes of memory\n",size);
    return(ptr);
}
```

source/Menu.c

```
/*
 * Pull-Right Menu functions
 */

#include <stdio.h>
#include <X11/IntrinsicP.h>
#include <X11/StringDefs.h>

#include <X11/Xaw/XawInit.h>
#include <X11/Xaw/SimpleMenP.h>
#include <X11/Xaw/CommandP.h>

static void prPopupMenu();
static void NotifyImage();
static void PrLeave();

void InitActions(app_con)

XtAppContext app_con;

{
    static XtActionsRec actions[] = {
        {"prPopupMenu", prPopupMenu},
        {"notifyImage", NotifyImage},
        {"prLeave", PrLeave},
    };

    XtAppAddActions(app_con.actions, XtNumber(actions));
```

- 253 -

```
}

static void prPopupMenu(w.event.params.num_params)

Widget w;
XEvent * event;
String * params;
Cardinal * num_params;

{
    Widget menu, temp;
    Arg arglist[2];
    Cardinal num_args;
    int menu_x, menu_y, menu_width, menu_height, button_width, button_height;
    Position button_x, button_y;

    if (*num_params != 1) {
        char error_buf[BUFSIZ];
        sprintf(error_buf, "prPopupMenu: %s.", "Illegal number of translation
arguments");
        XtAppWarning(XtWidgetToApplicationContext(w), error_buf);
        return;
    }
    temp = w;
    while(temp != NULL) {
        menu = XtNameToWidget(temp, params[0]);
        if (menu == NULL)
            temp = XtParent(temp);
        else
            break;
    }
}
```


- 254 -

```
if (menu == NULL) {
    char error_buf[BUFSIZ];
    sprintf(error_buf, "prPopupMenu: %s %s.",
            "Could not find menu widget named", params[0]);
    XtAppWarning(XtWidgetToApplicationContext(w), error_buf);
    return;
}
if (!XtIsRealized(menu))
    XtRealizeWidget(menu);

menu_width = menu->core.width + 2 * menu->core.border_width;
button_width = w->core.width + 2 * w->core.border_width;
button_height = w->core.height + 2 * w->core.border_width;

menu_height = menu->core.height + 2 * menu->core.border_width;

XtTranslateCoords(w, 0, 0, &button_x, &button_y);
menu_x = button_x;
menu_y = button_y + button_height;

if (menu_x < 0)
    menu_x = 0;
else {
    int scr_width = WidthOfScreen(XtScreen(menu));
    if (menu_x + menu_width > scr_width)
        menu_x = scr_width - menu_width;
}

if (menu_y < 0)
    menu_y = 0;
else {
    int scr_height = HeightOfScreen(XtScreen(menu));
```

- 255 -

```

if (menu_y + menu_height > scr_height)
    menu_y = scr_height - menu_height;
}

```

```

num_args = 0;
XtSetArg(arglist[num_args], XtNx, menu_x); num_args++;
XtSetArg(arglist[num_args], XtNy, menu_y); num_args++;
XtSetValues(menu, arglist, num_args);

```

```

XtPopupSpringLoaded(menu);

```

```

}

```

```

/*

```

```

static void

```

```

prRealize(w, mask, attrs)

```

```

Widget w;

```

```

Mask *mask;

```

```

XSetWindowAttributes *attrs;

```

```

{

```

```

    (*superclass->core_class.realize) (w, mask, attrs);

```

```

*/

```

```

/* We have a window now. Register a grab. */

```

```

/*

```

```

XGrabButton( XtDisplay(w), AnyButton, AnyModifier, XtWindow(w),
              TRUE, ButtonPressMask|ButtonReleaseMask,
              GrabModeAsync, GrabModeAsync, None, None );

```

```

}

```

```

*/

```

```

static void NotifyImage(w,event,params.num_params)

```

```

Widget      w;

```

```

XEvent      *event;

```

- 256 -

String *params;

Cardinal *num_params;

{

CommandWidget cbw=(CommandWidget)w;

if (cbw->command.set) XtCallCallbackList(w,cbw->command.callbacks,event);

}

static void PrLeave(w,event,params,num_params)

Widget w;

XEvent *event;

String *params;

Cardinal *num_params;

{

SimpleMenuWidget smw=(SimpleMenuWidget)w;

Dprintf("PrLeave\n");

}

- 257 -

source/Message.c

```
/*
 *   Message I/O Utility Routines
 */

#include    "../include/xwave.h"
#include    <varargs.h>

#define     MESS_ICONS      3

void  TextSize(msg)

Message    msg;

{
    int     i=-1, max_len=0;
    char    *text=msg->info.ptr;

    msg->rows=0;
    msg->cols=0;
    do {
        i++;
        if (text[i] == '\n' || text[i] == '\0') {
            if (msg->cols > max_len) max_len=msg->cols;
            msg->cols=0;
            msg->rows++;
        } else msg->cols++;
    } while (text[i] != '\0');
    if (i > 0) if (text[i-1] == '\n') msg->rows--;
```

- 258 -

```
    msg->cols = max_len;
}

Message      NewMessage(text.size)

char  *text;
int   size;

{
    Message      msg = (Message)MALLOC(sizeof(MessageRec));

    msg->shell = NULL;
    msg->widget = NULL;
    msg->info.firstPos = 0;
    if (! (msg->own_text = text == NULL)) msg->info.ptr = text;
    else {
        msg->info.ptr = (char *)MALLOC(size + 1);
        msg->info.ptr[0] = '\0';
    }
    msg->info.format = FMT8BIT;
    msg->info.length = 0;
    msg->rows = 0;
    msg->cols = 0;
    msg->size = size;
    msg->edit = XawTextEdit;
    return(msg);
}
```

```
void  CloseMessage(w, closure.call_data)
```

```
Widget      w;
caddr_t     closure.call_data;
```

- 259 -

```

{
    Message    msg = (Message)closure;

    Destroy(w, (caddr_t)msg->shell.NULL);
    if (msg->own_text) XtFree(msg->info.ptr);
    XtFree(msg);
}

```

```

void    MessageWindow(parent.msg, title, close, call)

```

```

Widget    parent;

```

```

Message    msg;

```

```

char    *title;

```

```

Boolean    close;

```

```

XtCallbackRec    call[];

```

```

{
    Widget    form, widgets[MESS_ICONS] = {NULL, NULL, NULL};
    FormItem    items[] = {
        {"msg_cancel", "cancel", 0, 0, FW_icon, NULL},
        {"msg_label", title, 1, 0, FW_label, NULL},
        {"msg_msg", NULL, 0, 2, FW_text, (String)msg},
    };
}

```

```

msg->edit = XawTextRead;

```

```

msg->shell = ShellWidget("msg", parent, parent == global->toplevel?SW_top:SW_below,
    NULL, NULL);

```

```

form = FormatWidget("msg_form", msg->shell);

```

```

FillForm(form, MESS_ICONS-(close?0:1), &items[close?0:1], &widgets[close?0:1], call);

```

```

XtPopup(msg->shell, XtGrabNone);

```

- 260 -

```
Mflush(msg);
}

void Mflush(msg)

Message    msg;

{
    if (global->batch == NULL && msg->widget != NULL) {
        Display    *dpy = XtDisplay(global->toplevel);
        int        i, lines = 0;
        Arg        args[1];

        for(i = msg->info.length-1; lines < msg->rows && i >= 0; i--)
            if (msg->info.ptr[i] == '\n' && i != msg->info.length-1) lines++;
        i++;
        if (msg->info.ptr[i] == '\n') i++;
        strcpy(msg->info.ptr, &msg->info.ptr[i]);
        msg->info.length = i;
        XtSetArg(args[0], XtNstring, msg->info.ptr);
        XSynchronize(dpy, True);
        XtSetValues(msg->widget, args, ONE);
        XSynchronize(dpy, False);
    }
}

void mprintf(msg, ap)

Message    msg;
va_list    ap;

{
```

- 261 -

```
char *format;

format=va_arg(ap,char *);
if (global->batch!=NULL) vprintf(format,ap);
else {
    char text[STRLEN];
    int i;

    vsprintf(text,format,ap);
    i=strlen(text)+msg->info.length-msg->size;
    if (i>0) {
        strcpy(msg->info.ptr,&msg->info.ptr[i]);
        msg->info.length-=i;
    }
    strcat(msg->info.ptr,text);
    msg->info.length+=strlen(text);
}
}

void Dprintf(va_list)

va_dcl

{
    va_list ap;

    if (global->debug) {
        char *format;

        va_start(ap);
        format=va_arg(ap,char *);
        vprintf(format,ap);
```


- 262 -

```
        va_end(ap);
    }
}

void Mprintf(va_alist)

va_dcl

{
    va_list    ap;
    Message    msg;

    va_start(ap);
    msg = va_arg(ap, Message);
    mprintf(msg, ap);
    va_end(ap);
}

void Eprintf(va_alist)

va_dcl

{
    va_list    ap;
    Message    msg;
    int        rows, cols;

    va_start(ap);
    msg = NewMessage(NULL, STRLEN);
    mprintf(msg, ap);
    if (global->batch == NULL) {
        X1CallbackRec    callbacks[] = {
```

- 263 -

```
        {CloseMessage,(caddr_t)msg},  
        {NULL,NULL},  
    };  
  
    TextSize(msg);  
    MessageWindow(global->toplevel,msg,"Xwave Error".True.callbacks);  
}  
va_end(ap);  
}
```

- 264 -

source/NameButton.c

```
/*
 * Supply MenuButton widget id to PullRightMenu button resource
 */
```

```
#include    "../include/xwave.h"
```

```
void NameButton(w, event, params, num_params)
```

```
Widget      w;
```

```
XEvent      *event;
```

```
String *params;
```

```
Cardinal     *num_params;
```

```
{
    MenuButtonWidget mbw = (MenuButtonWidget) w;
    Widget menu;
    Arg args[1];
    String name;
    XtSetArg(args[0], XtNmenuName, &name);
    XtGetValues(w, args, ONE);
    Dprintf("NameButton: looking for PRM %s\n", name);
    menu = FindWidget(name, w);
    if (menu != NULL) {
        Dprintf("NameButton: setting Menu Button\n");
        XtSetArg(args[0], XtNbutton, w);
        XtSetValues(menu, args, ONE);
    }
}
```

- 265 -

source/Palette.c

```
/*
 *   Palene re-mapping
 */

#include    "../include/xwave.h"

/*   Function Name:    ReMap
 *   Description:    Re-maps a pixel value to a new value via a mapping
 *   Arguments:    pixel - pixel value (0..max-1)
 *                  max - range of pixel values
 *                  map - palette to recode with
 *   Returns:    remapped pixel value
 */

int    ReMap(pixel.max,palette)

int    pixel.max;
Palette    palette;

{
    Map    map=palette->mappings;
    int    value=pixel;
    Boolean    inrange=False;

    while(map!=NULL && !inrange) {
        if (pixel>=map->start && pixel<=map->finish) {
            inrange=True;
            value=map->m*pixel+map->c;
        }
    }
}
```

- 266 -

```

    }
    map=map->next;
}
return(value < 0?0:value > =max?max-1:value);
}

```

```

/*  Function Name:    FindPalette
 *  Description:    Find a palette from a list given the index
 *  Arguments:    palette - the palette list
 *                index - the index number
 *  Returns:      the palette corresponding to the index
 */

```

```

Palette    FindPalette(palette,index)

```

```

Palette    palette;
int        index;

{
    while(index > 0 && palette->next!=NULL) {
        index--;
        palette=palette->next;
    }
    return(palette);
}

```

```

/*  Function Name:    ReOrderPalettes
 *  Description:    Reverse the order of the palette list
 *  Arguments:    start, finish - the start and finish of the re-ordered list
 *  Returns:    the palette list in the reverse order
 */

```

- 267 -

Palette ReOrderPalettes(start.finish)

Palette start. finish:

```
{  
    Palette list = finish->next;  
  
    if (list != NULL) {  
        finish->next = list->next;  
        list->next = start;  
        start = ReOrderPalettes(list.finish);  
    }  
    return(start);  
}
```

- 268 -

source/Parse.c

```
/*
 *   Parser for xwave input files: .elo
 */

#include    "../include/xwave.h"
#include    "../include/Gram.h"

void  Parse(path,file,ext)

String path, file, ext;

{
    char  file_name[STRLEN];

    sprintf(file_name,"%s%s/%s%s\0",global->home,path,file,ext);
    Dprintf("Parse: parsing file %s\n",file_name);
    if (NULL == (global->parse_fp=fopen(file_name,"r")))
        Eprintf("Parse: failed to open input file %s\n",file_name);
    else {
        sprintf(file_name,"%s%s\0",file,ext);
        global->parse_file=file_name;
        global->parse_token=ext;
        yyparse();
        fclose(global->parse_fp);
        Dprintf("Parse: finished with %s\n",file_name);
    }
}
```

- 269 -

```
void ParseCtrl(w,closure,call_data)
```

```
Widget      w;
```

```
caddr_t     closure, call_data:
```

```
{
    Parse(".",((XawListReturnStruct *)call_data)->string,(String)closure);
}
```

```
int ParseInput(fp)
```

```
FILE *fp;
```

```
{
    int num;

    if (global->parse_token!=NULL)
        if (global->parse_token[0]=='\0') {
            num=(int)'\n';
            global->parse_token=NULL;
        } else {
            num=(int)global->parse_token[0];
            global->parse_token++;
        }
    else if (EOF==(num=getc(global->parse_fp))) num=NULL;
    return(num);
}
```


- 270 -

source/Pop2.c

```
/*  
    Global callbacks for popping popups and assorted utilities  
*/
```

```
#include    "../include/xwave.h"
```

```
void  Destroy(w,closure,call_data)
```

```
Widget      w;
```

```
caddr_t      closure, call_data;
```

```
{  
    Widget      widget=(Widget)closure;  
  
    if (widget!=NULL) XtDestroyWidget(widget);  
}
```

```
void  Quit(w,closure,call_data)
```

```
Widget      w;
```

```
caddr_t      closure, call_data;
```

```
{  
    XtDestroyApplicationContext(global->app_con);  
    exit();  
}
```

```
void  Free(w,closure,call_data)
```

- 271 -

```

Widget      w;
caddr_t     closure, call_data;

{
    if (closure != NULL) XFree(closure);
}

Widget      FindWidget(name, current)

String name;
Widget      current;

{
    Widget    target = NULL;

    while(current != NULL) {
        target = XtNameToWidget(current, name);
        if (target == NULL) current = XtParent(current);
        else break;
    }
    if (target == NULL) {
        Eprintf("Can't find widget: %s\n", name);
        target = global->toplevel;
    }
    return(target);
}

#define      NA_ICONS 2

void  NA(w, closure, call_data)

Widget      w;

```

- 272 -

```
caddr_t      closure, call_data;

{
    Widget
    shell = ShellWidget("na_shell", (Widget)closure, SW_below, NULL, NULL),
        form = FormatWidget("na_form", shell), widgets[NA_ICONS];
    FormItem  items[] = {
        {"na_confirm", "confirm", 0, 0, FW_icon, NULL},
        {"na_label", "This function is not available", 0, 1, FW_label, NULL},
    };
    XtCallbackRec  callbacks[] = {
        {Destroy, (caddr_t)shell}, {NULL, NULL},
    };

    FillForm(form, NA_ICONS, items, widgets, callbacks);
    XtPopup(shell, XtGrabExclusive);
}

void SetSensitive(w, closure, call_data)

Widget      w;
caddr_t      closure, call_data;

{
    XtSetSensitive((Widget)closure, True);
}
```

- 273 -

source/Process.c

```
/*
 *   Call sub-processes
 */

#include      "../include/xwave.h"
#include      <signal.h>
#include      <sys/wait.h>
#include      <sys/time.h>
#include      <sys/resource.h>

/*      Function Name:      Fork
 *      Description:      Executes a file in a process and waits for termination
 *      Arguments:      argv - standard argv argument description
 *      Returns:      dead process id
 */

int      Fork(argv)

char      *argv[];

{
    int      pid;
    union wait      statusp;
    struct rusage      rusage;

    if (0 == (pid = fork())) {
        execvp(argv[0], argv);
        exit();
    }
}
```

- 274 -

```

    } else if (pid > 0) wait4(pid,&statusp,0,&rusage);
    return(pid);
}

/*  Function Name:      zropen
 *   Description:      Open a file (or .Z file) for reading
 *   Arguments:      file_name - name of the file to be read
 *                   pid - pointer to process id
 *   Returns:      file pointer
 */

```

```
FILE *zropen(file_name,pid)
```

```
char *file_name;
```

```
int *pid;
```

```

{
    char z_name[STRLEN];
    String zcat[] = {"zcat",z_name,NULL};
    FILE *fp;

    if (NULL == (fp = fopen(file_name,"r"))) {
        static int up[2];

        sprintf(z_name,"%s.Z",file_name);
        pipe(up);
        if (0 != (*pid = fork())) {
            Dprintf("Parent process started\n");
            close(up[1]);
            fp = fdopen(up[0],"r");
        } else {
            Dprintf("Running zcat on %s\n",zcat[1]);

```

- 275 -

```
        close(up[0]);
        dup2( up[1], 1 );
        close( up[1] );
        execvp(zcat[0],zcat);
    }
}
return(fp);
}
```

```
/*  Function Name:    zseek
 *
 *  Description:    Fast-forward thru file (fseek will not work on pipes)
 *
 *  Arguments:    fp - file pointer
 *
 *                  bytes - bytes to skip
 */
```

```
void  zseek(fp,bytes)
```

```
FILE  *fp;
int   bytes;

{
    char  scratch[1000];
    int   i;

    while(bytes > 0) {
        int  amount = bytes > 1000 ? 1000 : bytes;

        fread(scratch,sizeof(char),amount,fp);
        bytes -= amount;
    }
}
```

- 276 -

```
void  zclose(fp,pid)
```

```
FILE  *fp;
```

```
int   pid;
```

```
{
```

```
    union wait  statusp;
```

```
    struct rusage rusage;
```

```
    fclose(fp);
```

```
    if (pid!=0) wait4(pid,&statusp,0,&rusage);
```

```
}
```

- 277 -

source/PullRightMenu.c

#if (!defined(lint) && !defined(SABER))

static char Xrcsid[] = "\$XConsortium: PullRightMenu.c,v 1.32 89/12/11 15:01:50 kit

Exp \$";

#endif

/*

* Copyright 1989 Massachusetts Institute of Technology

*

- * Permission to use, copy, modify, distribute, and sell this software and its
- * documentation for any purpose is hereby granted without fee, provided that
- * the above copyright notice appear in all copies and that both that
- * copyright notice and this permission notice appear in supporting
- * documentation, and that the name of M.I.T. not be used in advertising or
- * publicity pertaining to distribution of the software without specific,
- * written prior permission. M.I.T. makes no representations about the
- * suitability of this software for any purpose. It is provided "as is"
- * without express or implied warranty.

*

* M.I.T. DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE.
INCLUDING ALL

* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO
EVENT SHALL M.I.T.

* BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES
OR ANY DAMAGES

* WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION

* OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN

- 278 -

* CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

*/

/*

* PullRightMenu.c - Source code file for PullRightMenu widget.

*

*/

#include <stdio.h>

#include <X11/IntrinsicP.h>

#include <X11/StringDefs.h>

#include <X11/Xaw/XawInit.h>

#include <X11/Xaw/SimpleMenP.h>

#include "PullRightMenuP.h"

#include <X11/Xaw/SmeBSB.h>

#include "SmeBSBpr.h"

#include <X11/Xaw/Cardinals.h>

#include <X11/Xmu/Iter.h>

#include <X11/Xmu/CharSet.h>

#define streq(a, b) (strcmp((a), (b)) == 0)

#define offset(field) XtOffset(PullRightMenuWidget, simple_menu.field)

static XtResource resources[] = {

/*

* Label Resources.

*/

- 279 -

```
{XtNlabel, XtCLabel, XtRString, sizeof(String),
  offset(label_string), XtRString, NULL},
{XtNlabelClass, XtCLabelClass, XtRPointer, sizeof(WidgetClass),
  offset(label_class), XtRImmediate, (caddr_t) NULL},
```

/*

* Layout Resources.

*/

```
{XtNrowHeight, XtCRowHeight, XtRDimension, sizeof(Dimension),
  offset(row_height), XtRImmediate, (caddr_t) 0},
{XtNtopMargin, XtCVerticalMargins, XtRDimension, sizeof(Dimension),
  offset(top_margin), XtRImmediate, (caddr_t) 0},
{XtNbottomMargin, XtCVerticalMargins, XtRDimension, sizeof(Dimension),
  offset(bottom_margin), XtRImmediate, (caddr_t) 0},
```

/*

* Misc. Resources

*/

```
{ XtNallowShellResize, XtCAallowShellResize, XtRBoolean, sizeof(Boolean),
  XtOffset(SimpleMenuWidget, shell.allow_shell_resize),
  XtRImmediate, (XtPointer) TRUE },
{XtNcursor, XtCCursor, XtRCursor, sizeof(Cursor),
  offset(cursor), XtRImmediate, (caddr_t) None},
{XtNmenuOnScreen, XtCMenuOnScreen, XtRBoolean, sizeof(Boolean),
  offset(menu_on_screen), XtRImmediate, (caddr_t) TRUE},
{XtNpopupOnEntry, XtCPopupOnEntry, XtRWidget, sizeof(Widget),
  offset(popup_entry), XtRWidget, NULL},
{XtNbackingStore, XtCBackingStore, XtRBackingStore, sizeof(int),
  offset(backing_store),
  XtRImmediate, (caddr_t) (Always + WhenMapped + NotUseful)},
```

- 280 -

```
{XtNbutton, XtCWidget, XtRWidget, sizeof(Widget),
    offset(button), XtRWidget, (XtPointer)NULL},
};
```

```
#undef offset
```

```
static char defaultTranslations[] =
```

```
" <EnterWindow>:  highlight()  \n\
  <LeaveWindow>:   pull()       \n\
  <BtnMotion>:    highlight()  \n\
  <BtnUp>:        execute()";
```

```
/*
```

```
 * Semi Public function definitions.
```

```
*/
```

```
static void Redisplay(), Realize(), Resize(), ChangeManaged();
```

```
static void Initialize(), ClassInitialize(), ClassPartInitialize();
```

```
static Boolean SetValues(), SetValuesHook();
```

```
static XtGeometryResult GeometryManager();
```

```
/*
```

```
 * Action Routine Definitions
```

```
*/
```

```
static void Highlight(), Unhighlight(), Pull(), Execute(), Notify(), PositionMenuAction();
```

```
/*
```

```
 * Private Function Definitions.
```

```
*/
```

```
static void MakeSetValuesRequest(), CreateLabel(), Layout();
```

```
static void AddPositionAction(), PositionMenu(), ChangeCursorOnGrab();
```

- 281 -

```
static Dimension GetMenuWidth(), GetMenuHeight();
static Widget FindMenu();
static SmeObject GetEventEntry();
```

```
static XtActionsRec actionsList[] =
{
    {"pull",          Pull},
    {"execute",       Execute},
    {"notify",        Notify},
    {"highlight",     Highlight},
    {"unhighlight",   Unhighlight},
};
```

```
CompositeClassExtensionRec pr_extension_rec = {
    /* next_extension */ NULL,
    /* record_type */     NULLQUARK,
    /* version */         XtCompositeExtensionVersion,
    /* record_size */     sizeof(CompositeClassExtensionRec),
    /* accepts_objects */ TRUE,
};
```

```
#define superclass (&overrideShellClassRec)
```

```
PullRightMenuClassRec pullRightMenuClassRec = {
    {
        /* superclass */ (WidgetClass) superclass,
        /* class_name */  "PullRightMenu",
        /* size */        sizeof(PullRightMenuRec),
        /* class_initialize */ ClassInitialize,
        /* class_part_initialize */ ClassPartInitialize,
        /* Class init'ed */ FALSE,
        /* initialize */    Initialize.
    }
```

- 282 -

```

/* initialize_hook */ NULL.
/* realize */ Realize.
/* actions */ actionsList.
/* num_actions */ XtNumber(actionsList).
/* resources */ resources.
/* resource_count */ XtNumber(resources).
/* xrm_class */ NULLQUARK.
/* compress_motion */ TRUE.
/* compress_exposure */ TRUE.
/* compress_enterleave*/ TRUE.
/* visible_interest */ FALSE.
/* destroy */ NULL.
/* resize */ Resize.
/* expose */ Redisplay.
/* set_values */ SetValues.
/* set_values_hook */ SetValuesHook.
/* set_values_almost */ XtInheritSetValuesAlmost.
/* get_values_hook */ NULL.
/* accept_focus */ NULL.
/* intrinsics version */ XtVersion.
/* callback offsets */ NULL.
/* tm_table */ defaultTranslations.
/* query_geometry */ NULL.
/* display_accelerator*/ NULL.
/* extension */ NULL
}, {
/* geometry_manager */ GeometryManager.
/* change_managed */ ChangeManaged.
/* insert_child */ XtInheritInsertChild.
/* delete_child */ XtInheritDeleteChild.
/* extension */ NULL
}, {

```

- 283 -

```

    /* Shell extension      */ NULL
}.{
    /* Override extension */ NULL
}.{
    /* Simple Menu extension*/ NULL
}
};

```

```
WidgetClass pullRightMenuWidgetClass = (WidgetClass)&pullRightMenuClassRec;
```

```

/*****
 *
 * Semi-Public Functions.
 *
 *****/

```

```

/*    Function Name: ClassInitialize
 *    Description: Class Initialize routine, called only once.
 *    Arguments: none.
 *    Returns: none.
 */

```

```
static void
```

```
ClassInitialize()
```

```

{
    XawInitializeWidgetSet();
    XtAddConverter( XtRString, XtRBackingStore, XmuCvtStringToBackingStore,
                   NULL, 0 );
    XmuAddInitializer( AddPositionAction, NULL);
}

```

```
/*    Function Name: ClassInitialize
```

- 284 -

- Description: Class Part Initialize routine, called for every
- subclass. Makes sure that the subclasses pick up
- the extension record.
- Arguments: wc - the widget class of the subclass.
- Returns: none.
- */

static void

ClassPartInitialize(wc)

WidgetClass wc;

{

SimpleMenuWidgetClass smwc = (SimpleMenuWidgetClass) wc;

/*

- Make sure that our subclass gets the extension rec too.

*/

pr_extension_rec.next_extension = smwc->composite_class.extension;

smwc->composite_class.extension = (caddr_t) &pr_extension_rec;

}

/* Function Name: Initialize

- Description: Initializes the simple menu widget
- Arguments: request - the widget requested by the argument list.
- new - the new widget with both resource and non
- resource values.
- Returns: none.
- */

/* ARGSUSED */

static void

Initialize(request, new)

- 285 -

```
Widget request, new;
{
    SimpleMenuWidget smw = (SimpleMenuWidget) new;

    XmuCallInitializers(XtWidgetToApplicationContext(new));

    if (smw->simple_menu.label_class == NULL)
        smw->simple_menu.label_class = smeBSBObjectClass;

    smw->simple_menu.label = NULL;
    smw->simple_menu.entry_set = NULL;
    smw->simple_menu.recursive_set_values = FALSE;

    if (smw->simple_menu.label_string != NULL)
        CreateLabel(new);

    smw->simple_menu.menu_width = TRUE;

    if (smw->core.width == 0) {
        smw->simple_menu.menu_width = FALSE;
        smw->core.width = GetMenuWidth(new, NULL);
    }

    smw->simple_menu.menu_height = TRUE;

    if (smw->core.height == 0) {
        smw->simple_menu.menu_height = FALSE;
        smw->core.height = GetMenuHeight(new);
    }

    /*
    * Add a popup_callback routine for changing the cursor.
```


- 286 -

*/

```
XtAddCallback(new, XtNpopupCallback, ChangeCursorOnGrab, NULL);
```

}

```
/* Function Name: Redisplay
```

```
* Description: Redisplays the contents of the widget.
```

```
* Arguments: w - the simple menu widget.
```

```
*          event - the X event that caused this redisplay.
```

```
*          region - the region the needs to be repainted.
```

```
* Returns: none.
```

*/

```
/* ARGSUSED */
```

```
static void
```

```
Redisplay(w, event, region)
```

```
Widget w;
```

```
XEvent * event;
```

```
Region region;
```

```
{
```

```
SimpleMenuWidget smw = (SimpleMenuWidget) w;
```

```
SmeObject * entry;
```

```
SmeObjectClass class;
```

```
if (region == NULL)
```

```
    XClearWindow(XtDisplay(w), XtWindow(w));
```

```
/*
```

```
* Check and Paint each of the entries - including the label.
```

```
*/
```

```
ForAllChildren(smw, entry) {
```

- 287 -

```

if (!XidsManaged ( (Widget) *entry)) continue;

if (region != NULL)
    switch(XRectInRegion(region, (int) (*entry)->rectangle.x,
        (int) (*entry)->rectangle.y,
        (unsigned int) (*entry)->rectangle.width,
        (unsigned int) (*entry)->rectangle.height)) {

        case RectangleIn:
        case RectanglePart:
            break;
        default:
            continue;
    }
class = (SmeObjectClass) (*entry)->object.widget_class;

if (class->rect_class.expose != NULL)
    (class->rect_class.expose)( (Widget) *entry, NULL, NULL);
}
}

/*  Function Name: Realize
 *  Description: Realizes the widget.
 *  Arguments: w - the simple menu widget.
 *              mask - value mask for the window to create.
 *              attrs - attributes for the window to create.
 *  Returns: none
 */

static void
Realize(w, mask, attrs)
Widget w;
XtValueMask * mask;

```

- 288 -

```

XSetWindowAttributes * attrs;
{
    SimpleMenuWidget smw = (SimpleMenuWidget) w;

    attrs->cursor = smw->simple_menu.cursor;
    *mask |= CWCursor;
    if ((smw->simple_menu.backing_store == Always) ||
        (smw->simple_menu.backing_store == NotUseful) ||
        (smw->simple_menu.backing_store == WhenMapped) ) {
        *mask |= CWBackingStore;
        attrs->backing_store = smw->simple_menu.backing_store;
    }
    else
        *mask &= ~CWBackingStore;

    (*superclass->core_class.realize) (w, mask, attrs);
}

```

```

/*    Function Name: Resize
 *    Description: Handle the menu being resized bigger.
 *    Arguments: w - the simple menu widget.
 *    Returns: none.
 */

```

```

static void
Resize(w)
Widget w;
{
    SimpleMenuWidget smw = (SimpleMenuWidget) w;
    SmeObject * entry;

    if ( !XtIsRealized(w) ) return;

```

- 289 -

```
ForAllChildren(smw, entry)    /* reset width of all entries. */
```

```
    if (XtIsManaged( (Widget) *entry))
```

```
        (*entry)->rectangle.width = smw->core.width;
```

```
    Redisplay(w, (XEvent *) NULL, (Region) NULL);
```

```
}
```

```
/* Function Name: SetValues
```

```
 * Description: Relayout the menu when one of the resources is changed.
```

```
 * Arguments: current - current state of the widget.
```

```
 *           request - what was requested.
```

```
 *           new - what the widget will become.
```

```
 * Returns: none
```

```
*/
```

```
/* ARGSUSED */
```

```
static Boolean
```

```
SetValues(current, request, new)
```

```
Widget current, request, new;
```

```
{
```

```
    SimpleMenuWidget smw_old = (SimpleMenuWidget) current;
```

```
    SimpleMenuWidget smw_new = (SimpleMenuWidget) new;
```

```
    Boolean ret_val = FALSE, layout = FALSE;
```

```
    if (!XtIsRealized(current)) return(FALSE);
```

```
    if (!smw_new->simple_menu.recursive_set_values) {
```

```
        if (smw_new->core.width != smw_old->core.width) {
```

```
            smw_new->simple_menu.menu_width = (smw_new->core.width != 0);
```

```
            layout = TRUE;
```

```
        }
```

```
        if (smw_new->core.height != smw_old->core.height) {
```

- 290 -

```

    smw_new->simple_menu.menu_height = (smw_new->core.height != 0);
    layout = TRUE;
}

}

if (smw_old->simple_menu.cursor != smw_new->simple_menu.cursor)
    XDefineCursor(XtDisplay(new),
        XtWindow(new), smw_new->simple_menu.cursor);

if (smw_old->simple_menu.label_string != smw_new->simple_menu.label_string)
    if (smw_new->simple_menu.label_string == NULL)        /* Destroy. */
        XtDestroyWidget(smw_old->simple_menu.label);
    else if (smw_old->simple_menu.label_string == NULL)    /* Create. */
        CreateLabel(new);
    else {
        /* Change. */
        Arg args[1];

        XtSetArg(args[0], XtNlabel, smw_new->simple_menu.label_string);
        XtSetValues(smw_new->simple_menu.label, args, ONE);
    }

if (smw_old->simple_menu.label_class != smw_new->simple_menu.label_class)
    XtAppWarning(XtWidgetToApplicationContext(new),
        "No Dynamic class change of the SimpleMenu Label.");

if ((smw_old->simple_menu.top_margin != smw_new->simple_menu.top_margin)
||
    (smw_old->simple_menu.bottom_margin !=
    smw_new->simple_menu.bottom_margin) /* filler..... */ ) {
    layout = TRUE;
    ret_val = TRUE;
}

```

- 291 -

```
if (layout)
```

```
    Layout(new, NULL, NULL);
```

```
return(ret_val);
```

```
}
```

```
/*  Function Name: SetValuesHook
```

```
 *   Description: To handle a special case, this is passed the
 *               actual arguments.
```

```
 *   Arguments: w - the menu widget.
```

```
 *               arglist - the argument list passed to XtSetValues.
```

```
 *               num_args - the number of args.
```

```
 *   Returns: none
```

```
*/
```

```
/*
```

```
 * If the user actually passed a width and height to the widget
```

```
 * then this MUST be used, rather than our newly calculated width and
```

```
 * height.
```

```
*/
```

```
static Boolean
```

```
SetValuesHook(w, arglist, num_args)
```

```
Widget w;
```

```
ArgList arglist;
```

```
Cardinal *num_args;
```

```
{
```

```
    register Cardinal i;
```

```
    Dimension width, height;
```

```
    width = w->core.width;
```

```
    height = w->core.height;
```

- 292 -

```

    for ( i = 0 ; i < *num_args ; i++ ) {
        if ( strcmp(arglist[i].name, XtNwidth) )
            width = (Dimension) arglist[i].value;
        if ( strcmp(arglist[i].name, XtNheight) )
            height = (Dimension) arglist[i].value;
    }

    if ((width != w->core.width) || (height != w->core.height))
        MakeSetValuesRequest(w, width, height);
    return(FALSE);
}

/*.....
 *
 * Geometry Management routines.
 *
 *.....*/

/*
 * Function Name: GeometryManager
 * Description: This is the SimpleMenu Widget's Geometry Manager.
 * Arguments: w - the Menu Entry making the request.
 *             request - requested new geometry.
 *             reply - the allowed geometry.
 * Returns: XtGeometry{Yes, No, Almost}.
 */

static XtGeometryResult
GeometryManager(w, request, reply)
Widget w;
XtWidgetGeometry * request, * reply;
{

```

- 293 -

```
SimpleMenuWidget smw = (SimpleMenuWidget) XtParent(w);
```

```
SmeObject entry = (SmeObject) w;
```

```
XtGeometryMask mode = request->request_mode;
```

```
XtGeometryResult answer;
```

```
Dimension old_height, old_width;
```

```
if ( !(mode & CWWidth) && !(mode & CWHeight) )
```

```
    return(XtGeometryNo);
```

```
reply->width = request->width;
```

```
reply->height = request->height;
```

```
old_width = entry->rectangle.width;
```

```
old_height = entry->rectangle.height;
```

```
Layout(w, &(reply->width), &(reply->height) );
```

```
/*
```

- * Since we are an override shell and have no parent there is no one to
- * ask to see if this geom change is okay, so I am just going to assume
- * we can do whatever we want. If you subclass be very careful with this
- * assumption, it could bite you.

```
*
```

- * Chris D. Peterson - Sept. 1989.

```
*/
```

```
if ( (reply->width == request->width) &&
      (reply->height == request->height) ) {
```

```
    if ( mode & XtCWQueryOnly ) { /* Actually perform the layout. */
```

```
        entry->rectangle.width = old_width;
```

```
        entry->rectangle.height = old_height;
```


- 294 -

```

    }
    else {
        Layout(( Widget) smw, NULL, NULL);
    }
    answer = XtGeometryDone;
}
else {
    entry->rectangle.width = old_width;
    entry->rectangle.height = old_height;

    if ( ((reply->width == request->width) && !(mode & CWHeight)) ||
        ((reply->height == request->height) && !(mode & CWWidth)) ||
        ((reply->width == request->width) &&
         (reply->height == request->height)) )
        answer = XtGeometryNo;
    else {
        answer = XtGeometryAlmost;
        reply->request_mode = 0;
        if (reply->width != request->width)
            reply->request_mode |= CWWidth;
        if (reply->height != request->height)
            reply->request_mode |= CWHeight;
    }
}
return(answer);
}

/*  Function Name: ChangeManaged
 *  Description: called whenever a new child is managed.
 *  Arguments: w - the simple menu widget.
 *  Returns: none.
 */

```

- 295 -

static void

ChangeManaged(w)

Widget w;

{

Layout(w, NULL, NULL);

}

/*.....

*

* Global Action Routines.

*

* These actions routines will be added to the application's

* global action list.

*

.....*/

/* Function Name: PositionMenuAction

* Description: Positions the simple menu widget.

* Arguments: w - a widget (no the simple menu widget.)

* event - the event that caused this action.

* params, num_params - parameters passed to the routine.

* we expect the name of the menu here.

* Returns: none

*/

/* ARGSUSED */

static void

PositionMenuAction(w, event, params, num_params)

Widget w;

XEvent * event;

String * params;

Cardinal * num_params;

- 296 -

```
{
Widget menu;
XPoint loc;

if (*num_params != 1) {
    char error_buf[BUFSIZ];
    sprintf(error_buf, "%s %s",
            "Xaw - SimpleMenuWidget: position menu action expects only one",
            "parameter which is the name of the menu.");
    XtAppWarning(XtWidgetToApplicationContext(w), error_buf);
    return;
}

if ( (menu = FindMenu(w, params[0])) == NULL) {
    char error_buf[BUFSIZ];
    sprintf(error_buf, "%s '%s'",
            "Xaw - SimpleMenuWidget: could not find menu named: ", params[0]);
    XtAppWarning(XtWidgetToApplicationContext(w), error_buf);
    return;
}

switch (event->type) {
case ButtonPress:
case ButtonRelease:
    loc.x = event->xbutton.x_root;
    loc.y = event->xbutton.y_root;
    PositionMenu(menu, &loc);
    break;
case EnterNotify:
case LeaveNotify:
    loc.x = event->xcrossing.x_root;
    loc.y = event->xcrossing.y_root;
```

- 297 -

```

    PositionMenu(menu, &loc);
    break;
case MotionNotify:
    loc.x = event->xmotion.x_root;
    loc.y = event->xmotion.y_root;
    PositionMenu(menu, &loc);
    break;
default:
    PositionMenu(menu, NULL);
    break;
}
}

```

```

/*****
 *
 * Widget Action Routines.
 *
 *****/

```

```

/*  Function Name: Unhighlight
 *  Description: Unhighlights current entry.
 *  Arguments: w - the simple menu widget.
 *              event - the event that caused this action.
 *              params, num_params - ** NOT USED **
 *  Returns: none
 */

```

```

/* ARGSUSED */

```

```

static void

```

```

Unhighlight(w, event, params, num_params)

```

```

Widget w;

```

```

XEvent * event;

```

- 298 -

```

String * params;
Cardinal * num_params;
{
    SimpleMenuWidget smw = (SimpleMenuWidget) w;
    SmeObject entry = smw->simple_menu.entry_set;
    SmeObjectClass class;

    if ( entry == NULL) return;

    smw->simple_menu.entry_set = NULL;
    class = (SmeObjectClass) entry->object.widget_class;
    (class->sme_class.unhighlight) ( (Widget) entry);
}

```

```

/*  Function Name: Highlight
    *  Description: Highlights current entry.
    *  Arguments: w - the simple menu widget.
    *              event - the event that caused this action.
    *              params, num_params - ** NOT USED **
    *  Returns: none
    */

```

```

/* ARGSUSED */

```

```

static void

```

```

Highlight(w, event, params, num_params)

```

```

Widget w;

```

```

XEvent * event;

```

```

String * params;

```

```

Cardinal * num_params;

```

```

{
    SimpleMenuWidget smw = (SimpleMenuWidget) w;
    SmeObject entry;

```

- 299 -

```
SmeObjectClass class;

if ( !XtIsSensitive(w) ) return;

entry = GetEventEntry(w, event);

if (entry == smw->simple_menu.entry_set) return;

Unhighlight(w, event, params, num_params);

if (entry == NULL) return;

if ( !XtIsSensitive( (Widget) entry)) {
    smw->simple_menu.entry_set = NULL;
    return;
}

smw->simple_menu.entry_set = entry;
class = (SmeObjectClass) entry->object.widget_class;

(class->sme_class.highlight) ( (Widget) entry);
}

/*    Function Name: Notify
 *    Description: Notify user of current entry.
 *    Arguments: w - the simple menu widget.
 *               event - the event that caused this action.
 *               params, num_params - ** NOT USED **
 *    Returns: none
 */

/* ARGSUSED */
```

- 300 -

```

static void
Notify(w, event, params, num_params)
Widget w;
XEvent * event;
String * params;
Cardinal * num_params;
{
    SimpleMenuWidget smw = (SimpleMenuWidget) w;
    SmeObject entry = smw->simple_menu.entry_set;
    SmeObjectClass class;

    if ( (entry == NULL) || !XIsSensitive((Widget) entry) ) return;

    class = (SmeObjectClass) entry->object.widget_class;
    (class->sme_class.notify)( (Widget) entry );
}

/*  Function Name: Pull
 *
 *  Description: Determines action on basis of leave direction.
 *
 *  Arguments: w - the pull right menu widget.
 *
 *              event - the LeaveWindow event that caused this action.
 *
 *              params, num_params - ** NOT USED **
 *
 *  Returns: none
 */

```

```

static void Pull(w, event, params, num_params)

```

```

Widget      w;
XEvent      *event;
String *params;
Cardinal     *num_params;

```